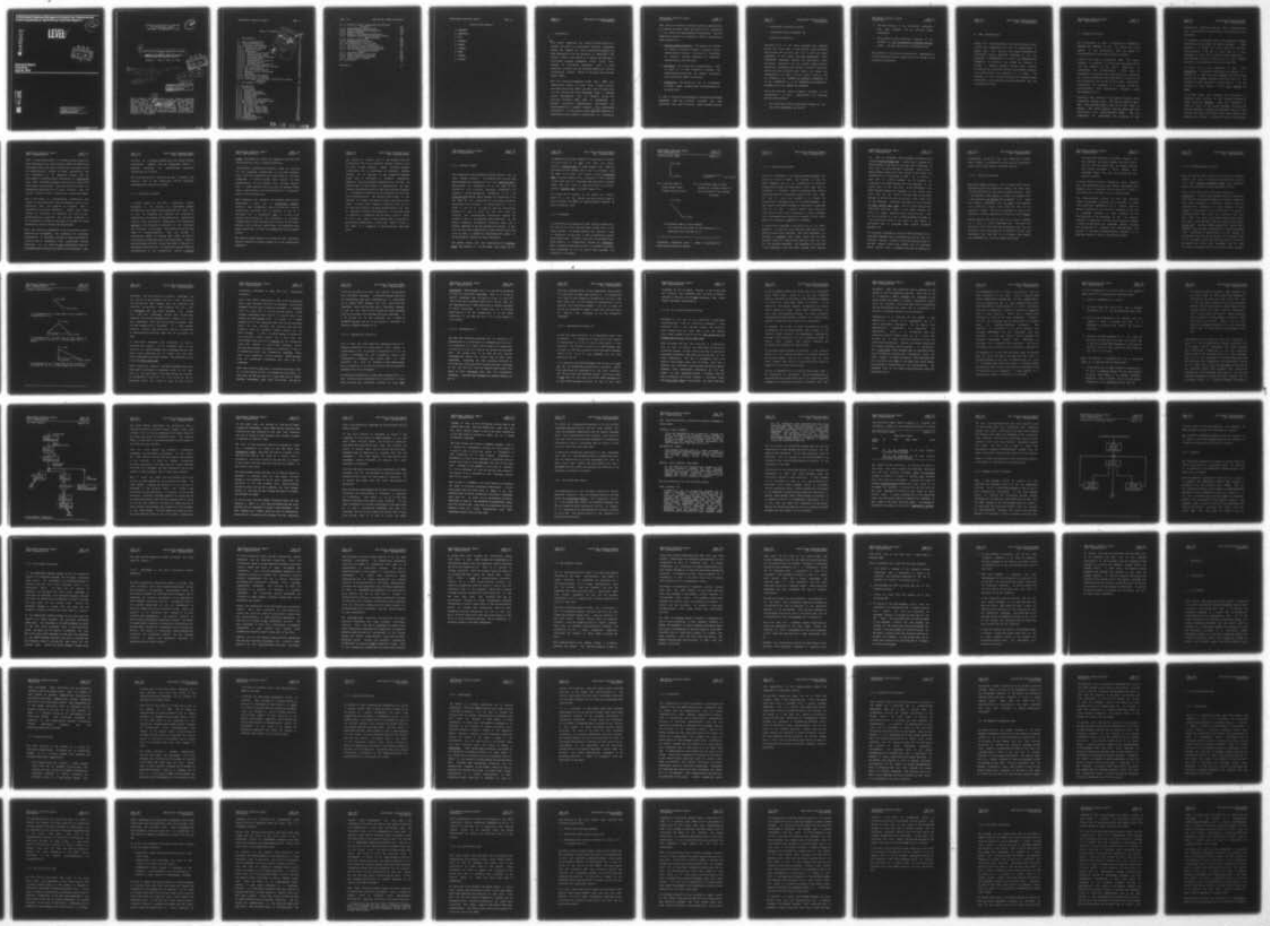
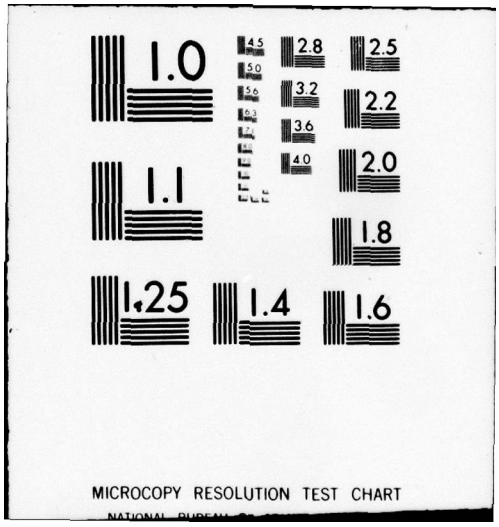
AD-A075 977 COMPUTER CORP OF AMERICA CAMBRIDGE MA F/G 5/2
A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND AND CONTRO--ETC(U)
JUL 79 N00039-77-C-0074
UNCLASSIFIED CCA-79-23 NL

1 OF 2
AD
A075977




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART

A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 5

LEVEL II

Technical Report
CCA-79-23
July 30, 1979

Computer Corporation of America
575 Technology Square

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

(12)

(6) A Distributed Database Management System for Command and Control Applications •
SEMI-ANNUAL TECHNICAL REPORT V

January 1, 1979 to June 30, 1979

(9) Semi-Annual technical rept. no. 5, 1 Jan - 30 Jun 79.

(11) 30 Jul 79

(14) CCA-79-23

(12) 168

(15) This research was supported by the Defense Advanced Research Project Agency of the Department of Defense and was monitored by the Naval Electronic System Command under Contract No. N00039-77-C-0074. ✓ The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

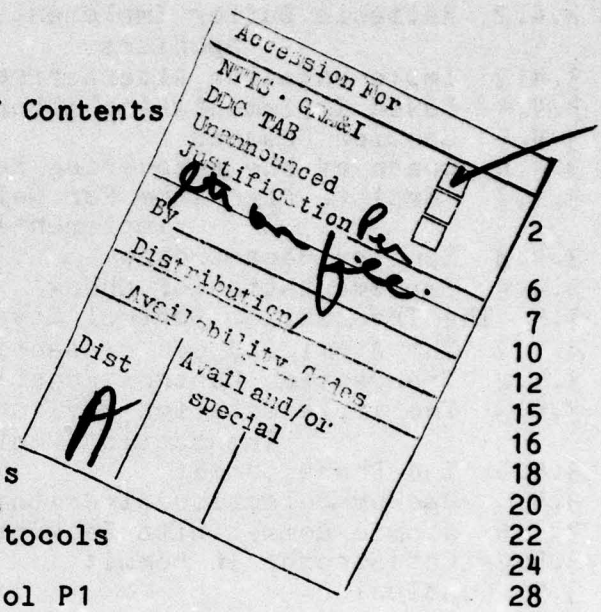


Table of Contents

Project Staff Members:

T. ANDERSON

P. BERNSTEIN

S. FOX

N. GOODMAN

M. HAMMER

T. LANDERS

C. REEVE

J. ROTHNIE

D. SHIPMAN

1. Introduction

This report summarizes the fifth six month period of a project entitled: "A Distributed Database Management System for Command and Control Applications" which has been undertaken by CCA and sponsored by ARPA-IPTO. The primary focus of this effort is to design and implement a distributed database management system called SDD-1 (System for Distributed Databases). SDD-1 is being installed in phases into the Advanced Command and Control Architectural Testbed (ACCAT) at the Naval Ocean Systems Center (NOSC).

SDD-1 is a database management system (abbr. DBMS) with capabilities similar to those found in many currently available centralized DBMSs. An SDD-1 user can store data into and retrieve data from the system using a language called Datalanguage [CCA e]. However, unlike any currently available DBMS, SDD-1 is implemented on multiple processors geographically distributed and interconnected in a computer network. This distributed architecture is motivated by three important benefits in applications which require a central pool of information

that must be accessed by multiple persons, organizations, or programs and where either the users of the information or its sources are geographically distributed. Military command and control is an example of such an application. These benefits are:

1. Reliability/Survivability - The system can continue to function correctly despite processor and/or communications failures. This level of robustness is achieved through redundancy of computers, communications lines and data.

2. Efficiency - By storing data geographically close to where it is most frequently accessed, the communications bottleneck and expense associated with centralized DBMSs is minimized.

3. Scalability - The system can grow to accommodate increased demand without major reconfiguration of existing sites.

In order to realize the advantages of distributed database management, some key technical problems have been identified that must be solved. These problems include:

- distributed concurrency control,
- distributed query processing, and
- achieving reliability.

Solutions to all of the above problems were designed during the first year of this project. These solutions have evolved over the last year and a half and major components of the system have been implemented. These implemented components include the distributed query processing capability and the update synchronization mechanisms. Throughout this project a series of technical reports have been produced that detail the design and implementation of SDD-1 [CCA a], [CCA b], [CCA c], [CCA d], [ROTHNIE et al], [ROTHNIE and GOODMAN], [BERNSTEIN et al a], [BERNSTEIN et al b], [BERNSTEIN and SHIPMAN a], [GOODMAN et al] and [HAMMER and SHIPMAN].

During the reporting period progress continued in the implementation of SDD-1. Specifically the following results were achieved:

1. The concurrency control algorithms designed by CCA have been implemented and tested.

2. The final version of the reliability mechanisms have been designed and are currently being implemented.

3. Three papers on SDD-1 have been submitted to and accepted by ACM Transactions on Database Systems (TODS). Two more papers will be submitted soon.

The remainder of this report describes the implementation of the concurrency control algorithm and the design of the reliability mechanisms.

2. SDD-1 Implementation

During this reporting period, the second working version of SDD-1 was implemented and will be demonstrated in August 1979. This version of the system implements the concurrency control algorithm developed by CCA and described in [BERNSTEIN et al a and b] and [BERNSTEIN and SHIPMAN a]. In addition, the system was enhanced to include a separate graphics process which visually illustrates the concurrency control and redundant updating as it occurs during the execution of an update. The remainder of this section describes the expanded system architecture, the concurrency control algorithm, and the new graphics process.

2.1 System Architecture

The architecture of SDD-1 is described in [ROTHNIE and GOODMAN] and [ROTHNIE et al]. We review here those aspects of the architecture that are needed for understanding the concurrency control mechanism.

A user of SDD-1 sees a conventional DBMS. The logical database is expressed in a relational data model which, from the viewpoint of the user's transaction, is nonredundant and nondistributed. Issues that are consequences of physical data distribution and redundancy are entirely handled by the system and are visible to the user transaction only insofar as they affect performance. Transactions are expressed as a program written in a semi-procedural data manipulation language called Datalanguage [CCA e].

Internally, SDD-1 consists of two types of modules, called transaction modules (abbr. TMs) and data modules (abbr. DMs). Each site can contain either one or both types of modules. DMs store physical data and behave much like conventional (i.e., nondistributed) DBMSs. TMs are responsible for supervising the execution of user

transactions, translating from the user's nondistributed view of the data to the realities of its distribution and redundancy.

For purposes of concurrency control the important messages processed by DMs are READ and WRITE messages. A READ message is a request by a TM to read some of the data items stored at a DM and to store them in a local workspace at that DM on behalf of some transaction. A WRITE message is sent by a TM to a DM to report updates produced by a transaction which the TM supervised.

The basic unit of user computation in SDD-1 is the transaction. A transaction essentially corresponds to a program in a high level host language with several data manipulation language statements sprinkled within it. The execution of each transaction is supervised by a TM and proceeds in three phases: called read, execute, and write.

In the read phase, SDD-1 analyzes the transaction to determine which portions of the (logical) database it reads, called its read-set. Since the transaction is coded in terms of the logical database, and since the physical database in general has redundant copies of many logical data items, the TM must choose which copies of the read-set will be read. It reads this copy of the read-set

into a private distributed workspace by sending READ messages to those DMs at which the selected copies are stored. A TM sends at most one READ message to each DM on behalf of a single transaction. If, for example, a transaction reads data from two data items that reside at the same DM, then only one READ message is issued to read both data items. This is an important point, as each DM performs READs and WRITEs as atomic operations. This means, for example, that none of the data read by a READ message can be updated by any WRITE during the time the READ is being processed. When all READ messages have been processed (i.e., when the TM has received positive acknowledgements from all the DMs), the read phase is complete.

During the execute phase, the TM supervises the execution of the transaction. This function of the TM is performed by the access planner and is described in [GOODMAN et al] and [WONG]. Since the concurrency control mechanism in the read phase guarantees that the physical read-set obtained by READ messages is internally consistent, the transaction will produce correct output. This output of this phase is produced in a workspace, not the permanent database.

In the write phase, the list of updates produced by the transaction is broadcast to the relevant DMs as WRITE messages. Each update to a logical data item, say x, is sent to all DMs that have a stored copy of x. The TM sends at most one WRITE message on behalf of a transaction to each DM.

2.2 Concurrent Correctness

The system usually has many transactions in progress at any one time, both because there are multiple TMs operating concurrently within the system and because individual TMs are processing transactions concurrently. If the READs and WRITEs that implement these transactions were arbitrarily interleaved, then serious problems of database consistency would result. The usual method of avoiding these consistency problems is by guaranteeing that the execution of transactions is serializable [ESWARAN et al] [PAPADIMITRIOU et al] [ROSENKRANTZ et al].

To guarantee serializability in SDD-1, we need to avoid undesirable interleavings of READ and WRITE messages -- those that lead to nonserializable executions. We accomplish this goal using two mechanisms. First, we examine each transaction to determine if it is conceivable

that it could participate in a nonserializable execution. Many transactions will never produce READs and WRITEs that interleave badly with other transactions, and hence can be run unsynchronized. Second, for those transactions that are determined to be dangerous because they can participate in nonserializable executions, we synchronize their READ and WRITE messages using protocols that avoid undesirable interleavings. These protocols are based on a timestamping mechanism and are quite different from the locking protocols used in conventional centralized DBMSs.

Most of the effort in distinguishing transactions that require no synchronization from the dangerous ones is done statically when the database is designed. When a transaction is actually submitted, a simple local table look-up is sufficient to determine how much, if any, synchronization is required. The run-time mechanism is the collection of protocols that must be invoked for those transactions that do require synchronization.

Note that these two components of the concurrency control mechanism are independent. Our technique for analyzing transactions to determine sources of nonserializability could be used in conjunction with conventional locking protocols. Or, we could run all transactions using our timestamp-based protocols and ignore the preanalysis step

entirely, as in present systems that use locking without preanalysis. Together the two mechanisms provide a powerful technique for synchronizing concurrent transactions at low cost.

The following sections describe two basic concepts that underlie much of the concurrency control mechanism, timestamps and transaction classes.

2.2.1 Transaction Classes

A crucial aspect of the SDD-1 concurrency control mechanism is its ability to distinguish between transactions that require synchronization and those that do not. By examining the read-set and write-set of transactions, the system can determine which transactions conflict with each other. Intuitively, two transactions conflict if the read-set or write-set of one intersects the write-set of the other. Such conflicts are the main cause of nonserializability. They are avoided in conventional DBMSs by locking data items so that two conflicting transactions never run concurrently. However, preventing all conflicts is more than what is required to guarantee serializability. By analyzing a graph theoretic representation of the transactions, called a conflict

graph, the system can isolate the dangerous conflicts that can potentially lead to nonserializability.

Unfortunately, analyzing the conflict graph at run-time for all executing transactions is too time consuming. Also, since the transactions are distributed at run-time, assembling a conflict graph would require too much communication. So, we transform this run-time analysis into a static analysis done only once at database design time by capitalizing on the predictability of transaction types in the following way.

When designing the database, the database administrator establishes a static set of transaction classes. Formally, each transaction class is defined by a logical read-set and write-set and is assigned to run at a particular TM. A transaction fits in a class if the read-set and write-set of the transaction is contained (respectively) in the read-set and write-set of the class. Read-set and write-set definitions are expressed using simple predicates, so that class membership can be checked quickly .

The conflict graph analysis is now done on the statically defined transaction classes instead of on the transactions themselves.

The utility of classes lies in the property that two transactions that run in different classes conflict only if their classes conflict. Hence, conflicts between transactions can be determined by conflicts between classes. So, an analysis of the classes at database design time is sufficient to determine potentially dangerous conflicts between transactions at run time. We believe that, for many kinds of applications, the most frequent determination will be that the class participates in no dangerous conflicts and can therefore run with only local synchronization. The conflict graph output of the analysis is a table, the protocol table, telling for each class: (a) which other classes it conflicts with, and (b) for each such conflict, how much synchronization (if any) is required to ensure serializability. At run-time, when a transaction is submitted, the TM finds a class in which it fits, and looks in the protocol table to see how to synchronize transactions in that class. What it finds in the table is a composite of the "protocols" described below.

2.2.2 Conflict Graphs

Two transactions from different classes conflict only if their classes conflict. To formalize this, we say that WRITE message for transaction T1 at a DM conflicts with a READ message for transaction T2 at the same DM iff transaction T1's write-set intersects transaction T2's read-set. A WRITE message for transaction T1 at a DM conflicts with another WRITE message for transaction T2 at the same DM iff transaction T1's write-set intersects transaction T2's write-set. It follows that if T1 is in class C1 and T2 is in class C2, then any intersection between T1's read or write set with T2's write set reflects the corresponding intersection between the classes C1 and C2. By examining class conflicts, we can predict potential transaction conflicts, which are a primary component of the serializability problem. It will turn out that this examination of class conflict will lead us tc our goal -- a method for determining the amount of synchronization required by each transaction.

The method begins with the construction of a conflict graph (see Figure 2.1). In the graph, each class, say Ci,

is modeled by two nodes labelled Ci's read and Ci's write. For each class, Ci, an edge <Ci's read, Ci's write>, called a vertical edge, is drawn (Figure 2.1a). When the write-sets of two classes, say C1 and C2, intersect, then an edge <C1's write, C2's write>, called a horizontal edge, is drawn (Figure 2.1b). Similarly, if the read-set of one class (say C1), intersects the write-set of another class (say C2), then an edge <C1's read, C2's write>, called a diagonal edge, is drawn (Figure 2.1c).

For a given set of classes, $\underline{C}$, we denote the conflict graph for $\underline{C}$ by $CG_{\underline{C}}$. We will use the conflict graph to help us predict the amount of synchronization required by each transaction class.

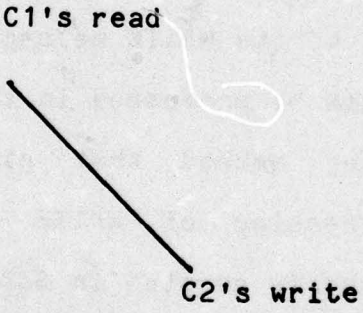
2.2.3 Timestamps

To synchronize two transactions that conflict dangerously, one must be run first, and the other delayed until it can safely proceed. In locking systems, the execution order is determined by the order in which transactions request conflicting locks. In SDD-1, the order is determined by a total ordering of transactions induced by timestamps. Each transaction executed by SDD-1 is assigned a globally unique timestamp by its TM before READ messages are broadcast on its behalf.

Conflict Graph Edges
Figure 2.1

(a) a vertical edge is drawn between every classes read and write.

(b) a horizontal edge is drawn between two classes writes iff the write-sets of the classes intersect.



(c)a diagonal edge is drawn between C1's read and C2's write iff the read-set of C1 intersects the write-set of C2.

Transaction timestamps serve a number of purposes for synchronizing READs and WRITEs.

2.2.3.1 Synchronizing Writes

One use of timestamps is in processing WRITE messages that arrive at a DM out of order. The problem is that the WRITE messages sent by two transactions in different classes which update the same logical data item may be processed in different orders at different DMs, thereby producing mutually inconsistent copies of the data item. One way to solve this problem is to attach the transaction's timestamp to all of its WRITE messages, and then require that WRITE messages be processed in timestamp order at all DMs. A better method that gives more flexibility to DMs in the processing of WRITE messages uses timestamped data items and is adopted in SDD-1 (this method was originally suggested in [THOMAS]).

A transaction's timestamp is carried on all of its WRITE messages. In addition, every physical data item at every DM has an associated timestamp. Note that timestamps are attached to physical data items; there may be many physical copies of a logical data item and each one has its own attached timestamp. The timestamp of a data item is the timestamp of the last WRITE message that updated

it. Each DM processes WRITE messages according to the following WRITE message rule: A data item is updated by a WRITE message if and only if the data item's timestamp is less than the WRITE message's timestamp. (Recall that a WRITE message contains the final values of data items, not computations to be performed on them.) So, to process a data item in a WRITE message, the DM compares the timestamp of the WRITE message with the timestamp of its stored copy of the data item. If the timestamp of the WRITE message exceeds the timestamp of the stored data item, then the new value of the data item in the WRITE message is written into the stored data item along with the new timestamp. Otherwise, the update is not performed on that stored data item. This is a data item by data item check; some data items in the WRITE message may result in update operations while others may not. That is, the net effect of a set of WRITE messages processed at a DM in arbitrary order is the same as the effect of processing them in timestamp order without the WRITE message rule.

The principal advantage of using the WRITE message rule is that WRITE messages can be processed as soon as they are received, thereby avoiding artificial queuing delays at the DMs. However, since later WRITEs may be processed before earlier ones, a database copy may be temporarily

inconsistent. As we will see, the concurrency control never permits a transaction to read such an inconsistent state if this could lead to incorrect results.

2.2.3.2 Synchronizing Reads

Each Read command contains a list of classes that conflict dangerously with the transaction issuing the Read (this list was determined by the conflict graph analysis). When a DM receives a Read command, it defers the command until it has processed all earlier Write commands (i.e., those with smaller timestamps) and no later Write commands (i.e., those with larger ones) from the TMs for the specified classes. The DM can determine how long to wait because of a DM-TM communication discipline called piping.

Piping requires that each TM send all its Write commands for a class to DMs in timestamp order. Thus when a DM receives a Write from (say) TM_X timestamped (say) TS_X, the DM knows it has received all Write commands from TM_X with timestamps less than TS_X. So, to process a Read command with timestamp TS_R, the DM proceeds as follows:

For each class specified in the Read command, the DM processes all Write commands from that class's TM up to (but not beyond) TS_R. If, however, the DM has already processed a Write command with timestamp beyond TS_R from one of these TMs, the Read is rejected.

To avoid excessive delays in waiting for Write commands, idle TMs periodically send null (empty) timestamped Write commands; also, an impatient DM can explicitly request a null Write from a TM that is slow in sending them.

The synchronization protocol we have just described roughly corresponds to locking, and is designed to avoid "race conditions" [BERNSTEIN et al c]. However, there are several variations of this protocol, depending on the type of timestamp attached to Read commands and the interpretation of the timestamps by DMs. For example, read-only transactions can use a less expensive protocol in which the DM selects the timestamp, thereby avoiding the possibility of rejection and reducing delay. The variety of available synchronization protocols is an important feature of SDD-1's concurrency control.

2.2.3.3 Timestamp Based Protocols

While the analysis that leads to the protocols is somewhat complex, the rules for selecting the protocols are not. It is these Protocol Selection Rules that completely govern the concurrency control mechanism of SDD-1.

First, let us state each of the three protocols.

Protocol P1: Prevents READ messages from one transaction that conflict with WRITE messages from another transaction from being processed in different relative orders at different DM's. Transaction T1 obeys protocol P1 with respect to transaction T2 if for each DM if T1's write is processed before (resp. after) and conflicts with T2's read, then T1's write is processed before (resp. after) T2's read at every DM where they both appear and conflict.

Protocol P2: Prevents a READ message from seeing WRITE messages from two other transactions in reverse timestamp order. Transaction T1 obeys protocol P2 with respect to transactions T2 and T3 if for any DM, if T1's read at that DM is processed before and conflicts with T2's write at the same DM and T3 has a later timestamp than T2, then T1's read is processed before T3's write at every DM where

they both appear and conflict; and if T1's read at that DM is processed after and conflicts with T2's write at the same DM and T2 has a later timestamp than T3, then T1's read is processed after T3's write at every DM where they both appear and conflict.

Protocol P3: Prevents race conditions. Transaction T1 obeys protocol P3 with respect to transaction T2 if for each DM where T1 reads and T2 writes, T1's read and T2's write are processed in timestamp order at every DM where they both appear and conflict.

The Protocol selection rules state which protocols should be invoked by which transactions. They are:

I. For all classes C1 and C2 such that C1's read set intersects with C2's write set for each pair of transactions T1 and T2 in C1 and C2 (respectively), T1 must obey protocol P1 with respect to T2 (see Figure 2.2(a)).

II. For each cycle in the conflict graph that contains a vertical edge, the following hold:

a. for all distinct classes C1, C2, C3, if edges <C1's read, C2's write> and <C1's read, C3's write> lie on the cycle, then for each set of transactions T1, T2, and T3 in C1, C2, and C3 (respectively), T1 must obey P2 with respect to T2 and T3 (see Figure 2.2(b)); and

b. for all distinct classes C1 and C2 such that <C1's read, C1's write> and <C1's read, C2's write> lie on the cycle, then for each pair of transactions T1 and T2 in C1 and C2 (respectively), T1 must obey P3 with respect to T2 (see Figure 2.2(c)).

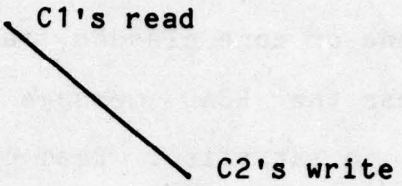
The protocol selection rules are easily transformed into an algorithm that analyzes the conflict graph and produces the protocols that each class must obey. However, the definitions of the protocols are not algorithmic. To make the protocols effective, we now show how TMs and DMs can enforce the relative orderings of READ and WRITE messages required by the protocols.

2.2.3.4 Read Conditions

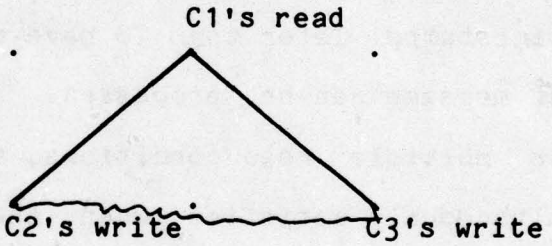
Each protocol demands that certain relative orderings of READ and WRITE messages be obeyed. These orderings are enforced by synchronization information that is carried entirely by the READ messages from a class in the form of read conditions.

A read condition is attached to a READ message and specifies which WRITE messages from certain other classes must be processed before the READ message can be correctly

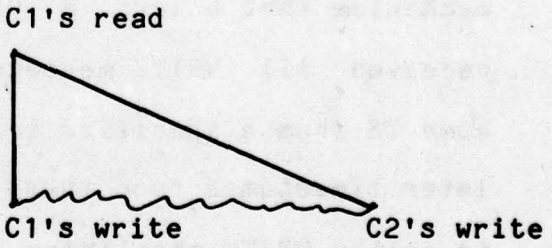
Protocol Selection Rules
Figure 2.2



(a) Transactions in C1 must obey P1 with respect to transactions in C2.



(b) Transactions in C1 must obey P2 with respect to transactions in C2 and C3, if these edges lie on a cycle.



(c) Transactions in C1 must obey P3 with respect to transactions in C2, if these edges lie on a cycle.

processed. The read condition includes a timestamp, say TS, and one or more classes, say {Cj,...,Cm}. The DM can only process the READ message when the attached read condition is satisfied. Read condition <TS, {Cj,...,Cm}> is satisfied when all WRITE messages from classes {Cj,...,Cm} with timestamps earlier than TS have been processed and no WRITE messages from classes {Cj,...,Cm} with timestamps later than TS have been processed. Then the READ message can be processed. If a READ message contains multiple read conditions, then all of them must be simultaneously satisfied when the DM processes the READ.

To effectively implement read conditions, we need a mechanism that allows a DM to determine when it has received all WRITE messages with timestamps earlier than some TS from a specified set of classes and none with later timestamps from these classes. The mechanism we use is called WRITE pipelining.

WRITE pipelining requires that WRITE messages from each class must be processed in timestamp order at all DMs. That is, for each class C1, for each DM and for any pair of transactions T1 and T2 in C1, T1's write at DM is processed before T2's write at that DM only if T1's

transaction timestamp is less than T2's transaction timestamp.

Given that WRITE pipelining is used, a DM can determine when a read condition <TS,{C2}> is satisfied. Since WRITE messages from any given class are processed in timestamp order at every DM, as soon as the DM receives a WRITE message timestamped later than TS, it knows to hold it and process the READ message first. Of course, if a WRITE message from C2 with timestamp later than TS was processed before the read condition was received, then the read condition cannot be satisfied without backing out the WRITE message. In SDD-1, no WRITE message is backed out for concurrency control reasons. So, in this case, the READ message would have to be rejected and the originating class must resubmit it with a later timestamp. Notice that all READ messages on behalf of transaction T1 have to be resubmitted with a new transaction timestamp, since their read conditions are now obsolete. WRITE messages from the resubmitted transaction will carry the new timestamp.

When read conditions are used, a problem arises when class C2 is idle because it has no transactions to process. In this case, the DM will wait for a long time until a WRITE message timestamped later than TS arrives. One way to

solve this problem is to have idle classes periodically send NULLWRITE messages. A NULLWRITE message specifies the originating class and a timestamp and is interpreted as an empty WRITE message from that class with that timestamp. When a DM receives such a NULLWRITE message, it can be sure that it has received all WRITE messages from the indicated class through the given timestamp. If a DM chooses not to wait passively for a WRITE or NULLWRITE message from C2, it can request a NULLWRITE by sending a SENDNULL message to C2.

2.2.3.5 Implementing Protocol P1

Let us show how read conditions implement protocol P1. Suppose transactions in C1 must obey P1 with respect to transactions in C2. To process a transaction T1 in C1, we select a timestamp TS' (not necessarily equal to the transaction's timestamp) and require that one read condition <TS', {C2}> be attached to each READ message sent on behalf of T1 to each DM at which conflicting WRITE messages from C2 are processed.

An important optimization is used when transactions in C1 only conflict with transactions in C2 at one DM. In this case, we avoid read conditions entirely by using READ

pipelining: READ messages from C1 to the DM are processed in order of transaction timestamps. That is, if T1 has an earlier timestamp than T2 (both in class C1), then T1's read at the DM is processed before T2's read at that DM. The implementation of READ pipelining is exactly analogous to WRITE pipelining. If transactions in C1 use READ pipelining at a DM and transactions in C2 use WRITE pipelining at the same DM, then P1 of C1 with respect to C2 is obeyed.

2.2.3.6 Implementing P3

The same read condition mechanism that we described for implementing P1 is sufficient for implementing P3 as well. For transaction T1 to obey P3 with respect to transactions in class C2 at a DM, T1's read at the DM must be processed after all writes from C2 at that DM with earlier timestamps and before all writes from C2 with later timestamps. Attaching the read condition <TS,{C2}> to T1's read will force the DM to process the read according to P3; the DM will wait for exactly those writes from class C2 with timestamps less that T1's transaction timestamp. Rejected READ messages are handled exactly as per P1.

From this implementation, we see immediately that protocol P3 is strictly stronger than protocol P1. If transactions in C1 obey P3 with respect to transactions in C2 at a DM, then they obey P1 with respect to transactions in C2 at that DM. The difference between P1 and P3 is that P1 allows any timestamp to appear in the read condition while P3 requires that timestamp to be the transaction timestamp.

2.2.3.7 Implementing Protocol P2

As with the other protocols, P2 is implemented using read conditions. If transaction T1 must obey P2 with respect to transactions in C2 and C3, then it must attach a read condition <TS, {C2,C3}> to each of its READ messages that are sent to a DM that processes conflicting WRITE messages from C2 or C3. As in P1, any timestamp for the read condition will do.

if T1 conflicts with WRITE messages from C2 and C3 at only one DM, an interesting optimization is possible. Rather than specifying the timestamp TS in the read condition, the DM can select the timestamp itself. As long as there is some time, TS, such that all earlier WRITE messages and no later WRITE messages from both C2 and C3 have been

processed, P2 will be obeyed. However, if two or more DMs are involved, the timestamp must be fixed in advance, because all DMs must use the same timestamp; they cannot choose timestamps independently.

2.2.3.8 P4: A Cycle-breaking Protocol

Although P1, P2, and P3 are sufficient to guarantee serializability, from an efficiency standpoint these protocols have a very serious problem. The problem is that a single class can cause many cycles and thereby force many classes to use P2 and P3, even though very few transactions are ever run in that class.

While we expect that the vast majority of transactions that we wish to execute are predictable and belong to predefined classes, we still want to be able to execute an unexpected transaction that does not fit into any of our class definitions. One way to accomplish this is to define a "very large" class, call it Ctotal, that has a read-set and write-set that includes the entire logical database. Every conceivable transaction can fit into Ctotal, so this apparently solves the problem. But the cost is enormous, for Ctotal induces a two-class cycle with every other class in the system. So, every class has

to run P3 against Ctotal, and Ctotal has to run P3 against every other class. Since P3 is the most expensive protocol (measured by the delay of the transaction obeying it), this is an unfortunate state of affairs. It is especially unfortunate because transactions will rarely need to execute in Ctotal, since most transactions fit into other less expensive classes. So, Ctotal introduces considerable synchronization overhead for synchronizing against a class that will rarely run a transaction.

In general, any class in which transactions are only infrequently run, but which creates many cycles in the conflict graph, exhibits this phenomenon. Although the problem of proliferation of cycles is especially acute in Ctotal, other classes with smaller read-sets and write-sets may manifest the same problem.

To alleviate these problems we introduce a new protocol called P4, the purpose of which is to "break" cycles in the conflict graph. That is, if a class runs P4, then other classes that are in a cycle with the P4 class can behave as if the cycle did not exist.

One way to implement P4 is to shut off the system when a P4 transaction is introduced. No new transactions are processed and the system works until all outstanding WRITE messages from transactions already in progress have been

processed. When the system has finally quiesced, we can safely run the P4 transaction serially. After all of the P4 transaction's WRITE messages are processed, we can safely permit the system to process transactions again. Since the execution before and after the P4 transaction ran was serializable and since the P4 transaction ran serially, the entire execution is serializable.

Implementing P4 by shutting off the system -- even temporarily -- is likely to be unacceptable due to a severe performance degradation. We can improve this implementation considerably by exploiting two observations. First, a P4 transaction need only synchronize against classes that lie on a cycle that includes the P4 class, since only classes on cycles can cause nonserializability. Second, even these classes need not quiesce completely before running a P4 transaction. Only conflicting WRITE messages must be completed before the P4 transaction executes and subsequently allows the other classes to resume processing. WRITE messages that do not conflict with READs in the same cycle cannot affect the crdering of transactions in the serialization. and therefore they do not require synchronization under the definition of P4.

The implementation of P4 differs structurally from the other protocols in two ways. First, P4 requires some direct communication between TMs. By this communication, the TM supervising the P4 class requests that certain other TMs perform synchronization to avoid interfering with the P4 transaction. Second, P4 requires an augmented form of read condition. Recall that a standard read condition is a pair of the form <timestamp, {classes}>. For P4, the timestamp may be interpreted as a "minimum time", i.e., <mintime=timestamp, {classes}>. This condition is satisfied if all WRITE messages from {classes} timestamped less than "timestamp" have been processed. It does not require that no messages from {classes} timestamped greater than "timestamp" be processed (as in standard read conditions). The utility of mintime read conditions will be explained momentarily.

To implement P4, we use three additional types of messages that are sent from TMs to TMs (not from TMs to DMs). A P4-ALERT message is sent from a TM supervising a P4 class to a TM supervising some other class. A P4-ALERT message includes the name of the P4 class and the timestamp of the P4 transaction as its parameters. A class responds to a P4-ALERT with either a P4-ACCEPT or a P4-REJECT.

To run a transaction T1 in the P4 class C1 with respect to some cycle CYC, one performs the following steps:

1. Choose a timestamp for T1, say TS.

2. For every class that lies on CYC, send a message P4-ALERT(C1,TS) to the TM supervising that class.

3. Wait for the P4-ACCEPTs to be received from all classes to which a P4-ALERT was sent. If a P4-REJECT is received, then restart the protocol from step 1.

4. Construct the READ messages for T1. For each DM and class C2 such that <C1's read, C2's write> lies on CYC and C2 sends WRITE messages to a DM that can conflict with C1's read, attach the read condition <TS, {C2}> to T1's read.

When a TM receives a P4-ALERT(C1,TS) for a particular class, C2, it performs the following steps:

1. If the TM has run or begun running a transaction in C2 with a timestamp greater than TS, then respond to the TM supervising T1 by sending P4-REJECT. Otherwise, send P4-ACCEPT and do not run another transaction in C2 timestamped earlier than TS.

2. For each DM to which C2 sends a READ message and for each class C3 which sends WRITE messages to that DM and for which <C2's read, C3's write> lies on CYC, the first transaction in C2 with timestamp greater than TS which issues a READ message to that DM must attach the read condition <mintime=TS, {C3}> to the read from class C2. These conditions are in addition to those normally carried by reads from class C2 to that DM. (Note: Only do this step for the first transaction in C2 with timestamp later than TS which sends a READ message to that DM.)

The combination of P4-ALERT and the read conditions in part(4) of the P4 algorithm are enough to guarantee that P4 is obeyed. Part (4) guarantees that WRITE messages from conflicting transactions in CYC with timestamps earlier than T1 are processed before T1's READ messages. Part (2) of the algorithm for P4-ALERT guarantees that WRITE messages from transactions conflicting with classes on CYC other than C1 and with timestamps earlier than TS are processed before READ messages from conflicting transactions with timestamps later than TS. This ensures that the transactions sending the WRITE messages can be serialized before T1. P4-REJECT messages are needed in

case the first transaction in C2 with timestamp later than TS is already in progress, for then it is too late to attach the read condition required by part (2) of the P4-ALERT algorithm. The mintime read condition is sufficient because only conflicting transactions with earlier timestamps need to be processed before the first transaction in C2; later ones can be safely processed. Together, the rules guarantee that transactions with earlier timestamps can be serialized before T1 and those with later timestamps can be serialized after T1

2.2.4 The Class Handler

The Class Handler comprises the submodules of the TM which supervise the phases of transaction execution which impact concurrency control and redundant updating.

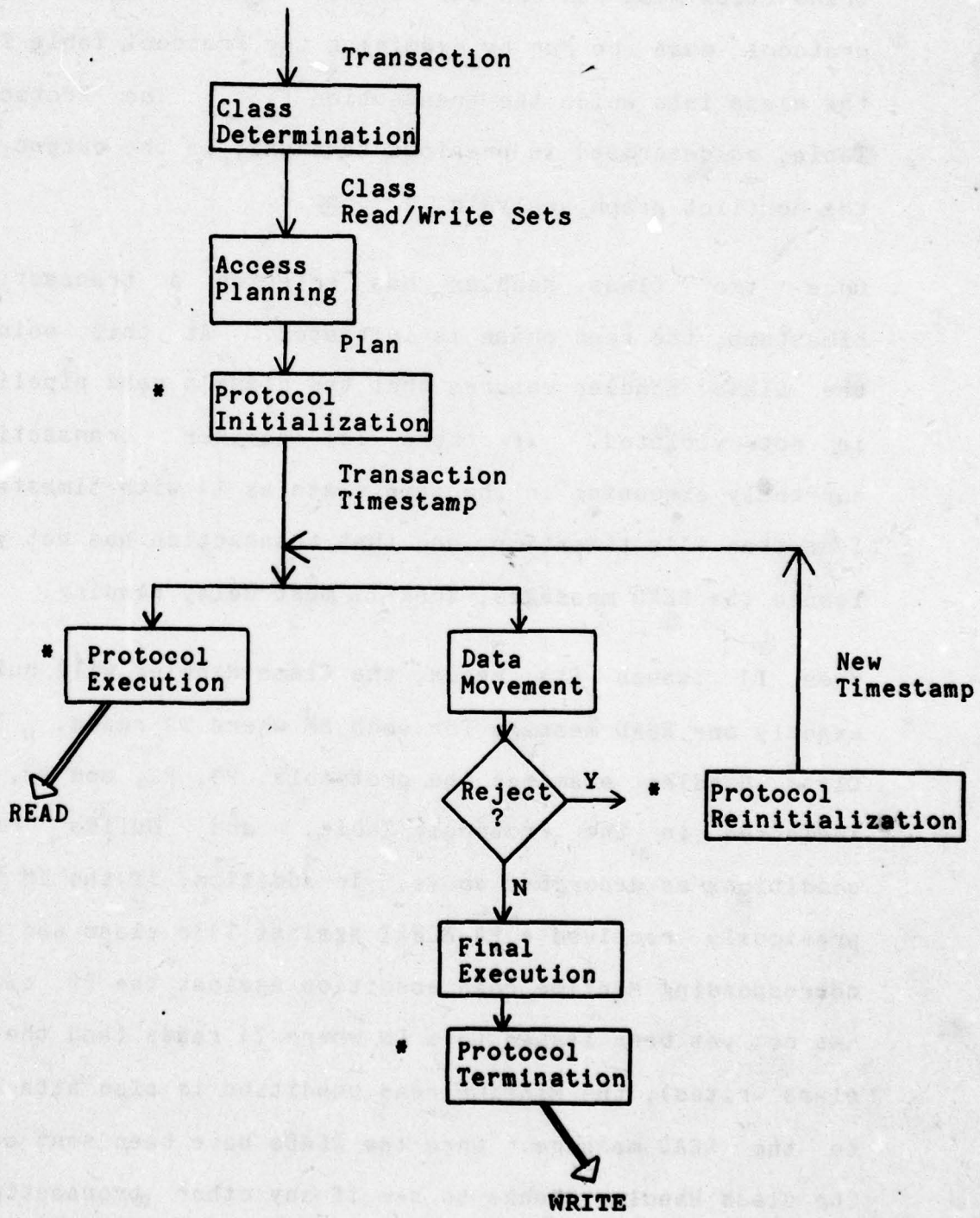
Since many transactions may be running concurrently at the same TM, more than one transaction in the same transaction class may be concurrently running. It is the responsibility of the Class Handler to keep track of the state of each transaction and enforce the read and write pipelines for each class, attach the appropriate read conditions to each READ issued for a transaction, and to ensure that all redundant copies of an updated data item receive a WRITE when the transaction terminates.

The Class Handler becomes actively involved during several steps of a transaction as illustrated in Figure 2.3.

Once the TM has transformed the user's transaction, T1, into a query or update in terms of logical fragments, it has determined T1's logical read and write sets. The class table is examined in order to determine into which class this transaction fits. The access planner then chooses the strategy for the execution phase, based on the locations of the stored fragments which will be read. At this point, the Class Handler is requested to select a transaction timestamp.

The Class Handler reads the TM's local clock (which is then incremented to ensure uniqueness of timestamps) and produces a transaction timestamp by appending the local site number (as the least significant digits) to the clock value. For most classes, this becomes the transaction timestamp. If T1 runs protocol P4, the Class Handler must first send a P4-ALERT to every TM who supervises a class which lies on a cycle with this P4 class. Once all TMs have accepted the P4-ALERT, the transaction is able to proceed. If any TM rejects the P4-ALERT (i.e., has already initiated a transaction with timestamp greater than the P4 timestamp), the Class Handler will select a new timestamp for T1 and reissue all of the P4-ALERTs.

The Class Handler
Figure 2.3



* Class Handler Submodules

The Class Handler determines the protocol(s) that a transaction must run and the classes against which each protocol must be run by examining the Protocol Table for the class into which the transaction fits. The Protocol Table, as described in previous sections, is the output of the conflict graph analysis.

Once the Class Handler has selected a transaction timestamp, the read phase is initiated. At this point, the Class Handler ensures that the class's read pipeline is not violated. If there is another transaction currently executing in the same class as T1 with timestamp less than T1's timestamp, and that transaction has not yet issued its READ messages, then T1 must delay reading.

When T1 issues its reads, the Class Handler will build exactly one READ message for each DM where T1 reads. The Class Handler examines the protocols, P1, P2, and P3, as indicated in the Protocol Table, and builds read conditions as described above. In addition, if the TM had previously received a P4-ALERT against T1's class and the corresponding Mintime read condition against the P4 class has not yet been issued to a DM where T1 reads (and the P4 class writes), the Mintime read condition is also attached to the READ message. Once the READs have been sent out, the Class Handler checks to see if any other transactions

in the same class are waiting to read due to larger transaction timestamps. Once a READ having a Mintime read condition has been accepted by a DM, that read condition will not be issued to that DM again until another P4-ALERT is received for that P4 class.

The list of read conditions which are attached to a READ for a transaction are based on the DMs where this transaction reads. Note that this may be a subset of the DMs which are read by the class as a whole, since a transaction fits into a class if its read set is a subset of the class's read set (and the write set is a subset of the class's write set).

The Data Movement phase executes, to as large an extent as possible, in parallel with the read phase. The Move Manager is the submodule of the TM which supervises the execution of the data movement plan. Before actually moving any data from one DM to another, the Move Manager checks to see that the DM, holding the data to be moved, has accepted the READ.

If, at any time, the Move Manager discovers that a DM has rejected a READ, it will stop processing and the Class Handler will be requested to select a new timestamp. The Move Manager will 'forget' about any data it successfully moved and will re-execute the strategy from the beginning

once a new transaction timestamp has been selected and new READs accepted.

If the Class Handler is requested to choose a new timestamp, it first sends out ABORT messages to all DMs where READs had been issued. This serves to release any data which had successfully been read (and permits any writes to execute which may be held up due this read). (Remember that all READs will be reissued with the new timestamp.) The Class Handler will then select a new timestamp, issuing P4-ALERTS and waiting for their responses if necessary.

Once the read phase has successfully terminated (all READs accepted) and all data movement has been performed, the transaction will enter its final phase of execution. It is during this phase that the final data values are determined.

When the DM has acknowledged the TM that the execution has terminated, the Class Handler is requested to terminate the transaction. Once again, it is up to the Class Handler to maintain the class's write pipeline. If any other transaction is currently executing in the same class as T1 with a transaction timestamp less than T1's timestamp, and has not yet issued its writes, then T1 must delay writing. Once T1 is able to write, one WRITE

message is sent to every DM having a stored copy of any updated data item. Once the WRITEs have been sent out, the Class Handler checks to see if any other transaction in the same class is waiting to WRITE due to a larger transaction timestamp.

The Class Handler may receive a SENDNULL message. This is a request for a NULLWRITE to be sent to a DM on behalf of a class in order to avoid long delays on acceptance of read conditions due to idle classes. Essentially a NULLWRITE for class CI at timestamp TI guarantees that all writes from class CI with timestamp less than TI have been sent. Included on the SENDNULL is the class of interest, CI, and a timestamp, TS, from the waiting read condition. The request is for a NULLWRITE at some timestamp greater than TS for class CI.

Upon receipt of a SENDNULL, the Class Handler will examine the state of currently executing transactions. It will also insure that its local clock is ahead of TS, thus ensuring that no future transaction could have a timestamp less than TS. If there are no transactions currently executing in class CI, the NULLWRITE is immediately issued with the current time. Note that this maintains the write pipeline since all future transactions will have timestamps greater than current time.

If there are transactions executing in CI, but all have timestamps greater than TS, then the Class Handler will issue the NULLWRITE immediately with the largest timestamp which still maintains the write pipeline. This will be a time just less than the minimum timestamp for transactions executing in class CI, indicating that all writes in class CI, up to that time, have been sent.

If there are transactions executing in CI with timestamps less than TS, the NULLWRITE cannot be immediately sent. Once all transactions in CI with timestamps less than TS have issued their WRITEs, the Class Handler will send a NULLWRITE and thus enable the waiting read condition to be satisfied as quickly as possible.

2.2.5 The Concurrency Module

The implementation of the run-time concurrency control mechanism primarily lies in a software module at the DMs called the Concurrency Monitor. The Concurrency Monitor at a DM accepts READ, WRITE, and NULLWRITE messages from TMs and schedules their execution at the DM. In essence, it is responsible for determining the ordering of events for local DMs. This section describes the operation of the Concurrency Monitor.

The Concurrency Monitor accepts and schedules messages of three types:

WRITE(TS, CLASS, UPDATES)

TS is the timestamp of the transaction issuing the WRITE, and CLASS is its transaction class. UPDATES is a list of data item identifiers and values. When a WRITE is processed, the indicated data items are updated to the specified values according to the WRITE Message Rule (see Section 2.2.3.1)

NULLWRITE(TS, CLASS)

This message indicates that all future messages in CLASS will have timestamp greater than TS. Processing the NULLWRITE simply involves taking note of this fact in the internal tables of the Concurrency Monitor.

READ(TS, CLASS, READSET, CONDITIONS)

TS and CLASS are the timestamp and transaction class of the transaction issuing the READ message. CONDITIONS is a list of read conditions associated with the READ message. Processing a READ involves reading the current values for data specified by READSET into a local transaction workspace.

The read conditions have the following format:

<TYPE, CLASSES, TS>

CLASSES is a list of transaction classes. TS is either a timestamp or is blank, depending on TYPE. If TYPE is "normal", then the read condition is satisfied when all WRITE messages from the listed classes with timestamps less than TS have been processed, but no WRITE messages from those classes with greater timestamps have been processed. "Normal" read conditions are used in all four protocols. If TYPE is "DMchoice", then the TS specification is blank; the read condition is satisfied when the condition for "normal" read conditions can be satisfied for some selected value

for TS. "DMchoice" read conditions are used in the single-DM optimized version of protocol P2. If type is "mintime", then the read condition is satisfied when all WRITE messages from the listed classes with timestamps less than TS have been processed. "Mintime" read conditions are used in the P4 protocol. The TS specification in a read condition must always be less than the transaction TS specified in the READ message itself (to prevent a deadlock within the concurrency monitor).

The DM returns an ACCEPT-READ message when all the read conditions on a READ message have been satisfied and the READ has been processed. If the read conditions cannot be satisfied, even by waiting for new WRITE messages to be processed, then a REJECT-READ message is returned to the originator of the READ.

The function of the Concurrency Monitor is to schedule the processing of READ and WRITE messages under the constraints imposed by read conditions. A READ message can be processed as soon as its read conditions are satisfied. While WRITE messages should be processed without unnecessary delay, a WRITE message will be delayed if its immediate processing would cause the rejection of a pending READ message. When a READ message is received, it is checked to see if it is immediately rejectable. If it is not, then the READ will eventually be satisfied, because the Concurrency Monitor will not process any WRITE messages that will cause it to be rejected.

The Concurrency Table, shown in Figure 2.4, contains the information needed by the Concurrency Monitor to resolve

---

The Concurrency Table Figure 2.4

Concurrency Table

Class	LW	LNW	Read Queue	Write Queue
i	.	.	.	.
	.	.	.	.

where:

LW is the timestamp of the most recently processed WRITE message.

LNW is the timestamp of the most recently processed NULLWRITE message.

---

the status of read conditions. It contains one entry for each class. Each entry contains the timestamp associated with the most recently processed write and nullwrite message; a pointer to a queue of pending read messages from that class to be processed; a pointer to a queue of pending write/nullwrite messages from that class to be processed and pointers to a hold list and wait list for the class. Messages are ordered on each queue by their arrival order. To avoid violating any of the pipelining rules, the Concurrency Monitor schedules the messages on each queue in the order that they appear. The message at the head of the queue is said to be immediately pending.

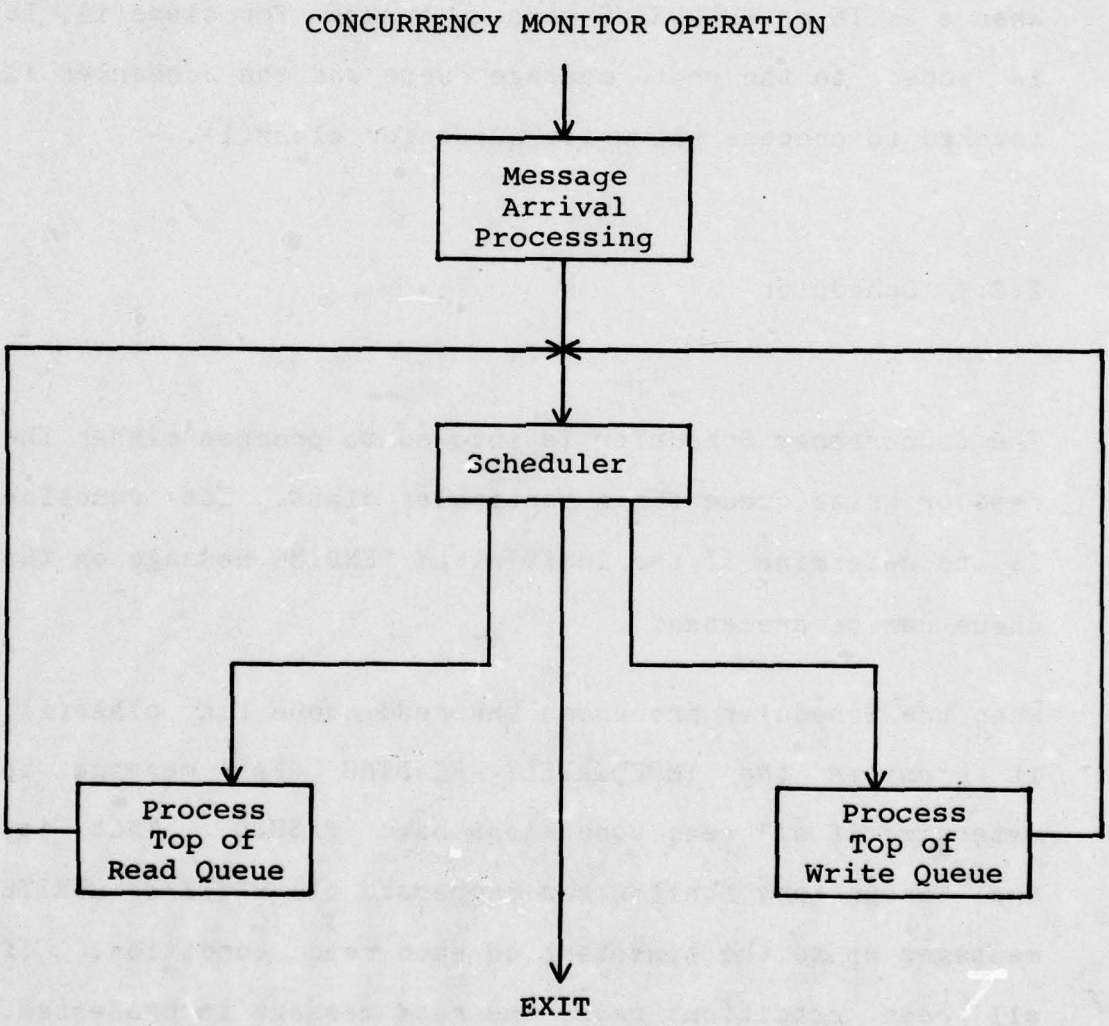
The Wait List associated with each class contains a list of other classes that have unsatisfied read conditions against the class. Each entry in the Wait List contains the waiting class number and timestamp of the unsatisfied read condition. The Hold List associated with each class contains a list of all read condition timestamps against the class. The Hold List is used to insure that processing a pending WRITE/NULLWRITE message does not cause a previously accepted read condition to be rejected.

Figure 2.5 shows the operation of the Concurrency Monitor. A description of each module is presented below.

2.2.6 Message Arrival Processing

When a READ message arrives for class(i) its read conditions are examined to see if it can be immediately rejected. If so, a REJECT-READ message is sent to the TM that originated the transaction. If the read message cannot be rejected the Concurrency Monitor chooses timestamps for any DMchoice read conditions and then adds the READ message to the read message queue for class(i). To insure that the Concurrency Monitor does not process any WRITE/NULLWRITE messages that would cause the READ message to be rejected, an entry is added to the Hold List

Operation of Concurrency Monitor
Figure 2.5



of every class in each read condition. The Scheduler is then invoked to process the read queue for class(i).

When a WRITE or NULLWRITE message arrives for class(i), it is added to the write message queue and the Scheduler is invoked to process the write queue for class(i).

2.2.7. Scheduler

The Concurrency Scheduler is invoked to process either the read or write queue for a particular class. Its function is to determine if the IMMEDIATELY PENDING message on the queue can be processed.

When the Scheduler processes the read queue for class(i), it examines the IMMEDIATELY PENDING READ message to determine if all read conditions have PASSED. That is, the Concurrency Monitor has processed all WRITE/NULLWRITE messages up to the timestamp on each read condition. If all read conditions pass, the read message is processed. However, if a read condition FAILs against class(j), the Scheduler adds an entry to the Wait List associated with class(j) and returns. As we will see later, this will insure that the read queue for class(i) will be

re-scheduled when a WRITE/NULLWRITE message in class(j) that satisfies the read condition is processed. Furthermore, to guarantee that the necessary WRITE/NULLWRITE message for class(j) arrives, the Concurrency Monitor may send a SENDNULL message to the TM that runs class(j) requesting it to send a NULLWRITE message to the DM.

When the Scheduler processes the write queue for class(i) it examines the IMMEDIATELY PENDING WRITE or NULLWRITE message. A NULLWRITE message is always processed immediately. However a WRITE message is only processed if there are no entries in class(i)'s Hold List with timestamps less than the timestamp on the WRITE message (i.e. processing the WRITE message would not cause previously accepted READ messages to be rejected).

2.2.8 Read Queue Processing

Processing a READ message for class(i) entails copying all data items in the READSET to a temporary work space. An ACCEPT-READ message is sent to the TM originating the transaction. Once this has been accomplished, the Hold List entries for each read condition are released and the Scheduler invoked to process the write queue for each class where a hold list entry was removed. This enables any WRITE messages blocked by this READ message to be processed. Finally, the READ message is popped from the queue and the Scheduler invoked to process the read queue for class(i).

2.2.9 Write Queue Processing

If the IMMEDIATELY PENDING message on the write queue for class(i) is a NULLWRITE message, the following processing takes place. The timestamp on the NULLWRITE message is stored in the last null write processed field of the Concurrency Table. The Wait List for class(i) is examined. For each Wait List entry whose timestamp is less than the timestamp on the NULLWRITE message, the Wait List entry is removed and the Scheduler invoked to process the read queue for the waiting class. Finally the NULLWRITE message is popped from the queue and the Scheduler invoked to process the write queue for class(i).

If the IMMEDIATELY PENDING message on the write queue for class(i) is a WRITE message, the following processing takes place. The timestamp on the WRITE message is stored in the last write processed field in the Concurrency Table. The Wait List for class(i) is examined. For each Wait List entry whose timestamp is less than the timestamp on the WRITE message, the Wait List entry is removed and the Scheduler called to process the read queue for the waiting class. Finally the WRITE message is popped from

the queue and the Scheduler invoked to process the write queue for class(i).

2.2.10 Advantages of the SDD-1 Concurrency Control Mechanism

The SDD-1 approach to concurrency control is in many ways quite different from other proposed mechanisms. We see many strengths in the approach. Unfortunately, there are few analytic methods for verifying these strengths, say by comparing the relative performance of our mechanism to other database concurrency controls. Furthermore, most of the proposed mechanisms are not yet implemented, so empirical comparisons are not possible either. Hence, the analysis of our mechanism must necessarily be more intuitive than mathematical. The specific criteria on which we base performance comparisons include: the amount of communication required to synchronize transactions; the average delay incurred by a transaction due to concurrency control; the amount of concurrency among transactions allowed by the concurrency control; and the overhead involved in making the mechanism resilient to communications and node failures.

At the architectural level, the SDD-1 concurrency control mechanism has two important properties. First, the architecture makes a strong separation between concurrency control issues and those of query processing and reliability. From a project management standpoint, this separation allowed us to attack the concurrency control problem independently from and in parallel with query processing and reliability problems. From a software engineering standpoint, this division of labor led naturally to a division of function in software components. The concurrency control mechanisms are isolated in a small number of modules, making them easily modifiable and tunable.

Second, the architecture fully distributes the concurrency control. While each transaction is controlled from a single site, different sites are concurrently supervising the synchronization of many different transactions. No one site is in charge of any system-wide activity. The main advantage of this full distribution is enhanced reliability. A site failure only affects those transactions executing and/or using data at that site.

However, it is in the specific synchronization mechanisms that the most important advantages lie: conflict graph analysis and the timestamp-based protocols. We believe

the technique of conflict graph analysis to be our most important contribution. By preanalyzing transaction conflicts, the number of transactions that need to be synchronized is drastically reduced. This has a beneficial effect on all aspects of concurrency control performance. It allows more concurrency among transactions; and for those transactions that require little or no synchronization it cuts delay, communications overhead, and costs associated with resiliency mechanisms. As shown in [BERNSTEIN and SHIPMAN b], the technique is quite general and can be used with a variety of synchronization protocols, including conventional locking. In principle, every proposed concurrency control mechanism could be improved by adding conflict graph analysis as a preprocessing step to eliminate run-time synchronization for some transactions.

The timestamp-based protocols, {P1,P2,P3,P4}, also offer important advantages over other proposed concurrency controls. First, the use of timestamps to resolve races among transactions eliminates the possibility of deadlock. Deadlock detection must be incorporated in any locking system and induces communications costs that the SDD-1 mechanism avoids. Second, the protocols synchronize transactions only against named transaction classes. Even if two transaction classes must be synchronized relative

to certain data, other classes can concurrently access that data; in fact, other classes can independently be synchronized against that very same data without affecting the first two classes at all. This is in contrast to locking protocols, which set blanket locks that apply to all transactions that access the shared data. Third, SDD-1 offers a range of synchronization protocols. Protocol P2 is a fast synchronization protocol for read-only transactions that can afford to read an old, but consistent copy of the database. While with a locking strategy read-only transactions could choose not to lock the data they read, that unlocked data may be inconsistent. Protocol P4 allows infrequently executed transactions to take a larger share of the synchronization burden. By running such transactions under P4, other frequently executed transactions can run P1 with less delay and more concurrency than they would obtain if they ran P2 or P3 as otherwise required. The P4 capability is currently unique to the SDD-1 mechanism.

2.3 The Graphics Process

One of the design goals of SDD-1 is to make distribution invisible to the user. Unfortunately, this makes it rather difficult to illustrate its operation as a distributed system. In its normal configuration, there are two choices: from one terminal, one will see something that looks very much like a centralized DBMS; from several terminals, one will see the system's distribution, but in a manner that is almost impossible to follow in real time.

The query processing aspect of SDD-1 was illustrated in part by a graphics program, running as part of a TM, that showed data movement among sites ..ough the execution of the final query. Several factors made this approach possible: since the system did not yet support updates, there were no possible synchronization problems; the controlling TM for a given transaction completely controlled the sequence of steps taken to answer the query.

With updating added to the system, though, a different approach was needed. The important aspects of SDD-1's

concurrency control mechanisms only come into play when several transactions are executing simultaneously, so the system must be able to illustrate this. If several transactions are executing, no single system module will have enough information to show the concurrency mechanisms operating: a transaction at site 1 may be affected by a transaction at site 4, but site 1 has no way of knowing that anything at all is happening at site 4.

The display is maintained by a separate process, running under MSG. The other modules in the system (TM's and DM's) send messages to the graphics process whenever an "interesting" event occurs. The graphics process then modifies its internal state, and possibly the display, to reflect the new situation. In contrast to the query processing demo, every system module must participate actively.

The demo of redundant update is similar in appearance to that of query processing. A color graphics terminal (a Ramtek Micrographic) is used to display information about four sites, corresponding to the four physical sites SDD-1 currently runs on. Each site's display is divided into three sections: a TM, a DM, and a data base window. The graphics process maps various events in SDD-1 onto the display, as follows:

Each site's TM has room for two items of data: the current timestamp at the TM, and some text describing what it's currently doing (For example, "DataMove" is displayed during the data movement phase of a transaction; "Class nn" is displayed when a transaction in class nn is initiated.) The timestamp is similar in form to, though much shorter than, SDD-1's actual timestamps: the low-order digit is a site identifier, and the three high-order digits are a sequence number. These short-form timestamps are used throughout the demo to identify transactions.

The DMs are divided into four sections, corresponding to the four sites. Thus, information regarding transactions originating at TM 1 will be displayed in the upper-left quadrant of any DM accessed. This includes responses to read messages from the TM, and an indication that the DM is executing the final datalanguage for a transaction.

Each site also has a database window, displaying the values and timestamps for six data items. Values are indicated by color; timestamps are like those displayed in TMs. Each data item also has a name associated with it.

Messages are illustrated by arrows from the sender to the receiver. The transaction timestamp is displayed with

each arrow, and it may also have a datum name or a protocol name.

Thus a transaction will cause the following sequence:

1. A TM sends a message to the graphics process indicating that a transaction in class n is starting. The displayed timestamp for the TM is updated, and 'Class n' is written into it.

2. Read messages are sent to various DMs, and to the graphics process.

3. Arrows are drawn from the sending TM to the receiving DMs.

4. As replies to the read messages arrive, they are displayed in the appropriate DMs. Possible replies are: 'queue', the read will eventually be accepted; 'accept', an accept has been sent to the TM; 'read', the requested data is being accessed in the database; 'refuse', the read message was refused by the concurrency control mechanism; and 'insuff', the read message was refused because the DM doesn't currently have the resources required to process the request. In the normal case, two or three read responses will be displayed in sequence for each read message.

5. If data movement is necessary, the TM will have 'DataMove' displayed in it while it's going on. The data movement is not shown in any detail, since the demonstration of query processing deals with it quite well.

6. When data movement is complete, the final datalanguage for the transaction is executed at one of the DMs. All read arrows are erased, along with the displayed responses. An arrow is drawn from the TM to the DM at the final site, and 'XCT' is displayed in the TM's quadrant.

7. If the transaction does any writing, the DM will send update messages to all other DMs where the data are stored. When these are sent, arrows are drawn from the sending DM to the receiving DM, with the transaction timestamp. The datum name in the database window is changed from white to the color of the sending DM, indicating that an update has been received but not yet installed.

8. As each DM installs the update, its database window is updated. During this phase, there will be two or more distinct values on the display for redundantly-stored data, but everything will eventually converge.

Of course, the above list describes only the ideal case. It is frequently the case, due to the physical distribution of the processes, that the response to some message will arrive at the graphics process before the message itself. In these cases, the graphics process remembers the response, and displays it only when the message appears. In addition, several restrictions are imposed by the limited size of the display. Although four transactions can be shown at a given DM, there can only be one transaction active between any TM-DM pair. If another transaction starts, the display for the earlier one will be erased before completion.

3. Reliability

3.1 Introduction

3.1.1 The RelNet

This section describes the Reliable Network, a subsystem of the SDD-1 distributed database management system, whose function it is to provide the level of reliability and robustness demanded of a system that is charged with responsibility for an organization's data. One of the prime motivations for building a distributed system is to achieve reliability enhanced over that which would be provided by a single-site system; the redundancy of data and processors provided by a distributed system potentially enable it to continue in operation despite the failure of individual sites. (In a single-site system, of course, site failure causes the entire system to cease

operation.) However, although a multi-site system presents opportunities for enhanced reliability, it also presents challenges in the same area, because the likelihood that some part of the total system will fail becomes much higher. The goal is to design and implement distributed systems that exhibit global robustness and toleration of local site failures, that can continue to operate as a whole despite the asynchronous failures and potential recoveries of individual elements of the system.

The Reliable Network (known as the RelNet) has been designed to provide a set of capabilities to support such reliable distributed system operation. Its original conception was in the context of the SDD-1 system, and some of its features were motivated by the particular requirements of that system. However, we believe that the functional capabilities of the RelNet have wide applicability to distributed systems of many kinds; in fact, we believe that it may represent a forerunner of a general reliable distributed operating system. This paper describes the functionality, architecture, and implementation techniques of the RelNet. The particular ways in which the RelNet is used to support reliable operation of SDD-1 are described in [ROTHNIE et al].

The RelNet consists of a set of facilities intended to ensure reliable communication and coordination among related processes operating at sites connected by means of a communications network. In a distributed system, a function will in general be realized by means of a number of processes, executing in parallel at distinct sites of a network; as execution of these processes proceeds, they will find occasion to communicate and synchronize with each other. The designer of a distributed system will have to recognize the reality that individual sites and processes in this system may fail at any point in time; consequently, each site must be prepared to recognize and react to the failure(s) of its "cohorts" [GRAY], the other sites with which it cooperates and interacts. One approach would be to embed this responsibility in the application logic and code of each cohort. However, following general principles of good software design, it would be preferable to provide the application programmer with a (fictional) view of the environment, which exhibits a degree of reliability that simplifies his system design and implementation. This view is the RelNet, a "virtual machine" network that may be used by the application system implementer as though it existed. The RelNet provides each process running in the system with a set of facilities for reliable communication and interaction with

other processes; these facilities can be utilized by invoking a set of procedure calls. Thus, the RelNet is used instead of whatever communication facilities are provided by the actual communication network connecting the sites in the distributed system. Many implementations of the RelNet facilities would be possible; we have chosen to realize it "on top" of the real network, by means of a software front-end on each machine in the distributed system. This front-end represents an interface to the communication network, intervening between application programs and the operating/communication system.

3.1.2 RelNet Facilities

The basic function of any network is to allow for inter-site communication. The RelNet can be effectively thought of as a virtual network that provides the following additional capabilities.

1. There exists within the network a single Global Clock that can be accessed from any site. The function of such a clock is to impose a uniform and consistent ordering on events occurring at different sites in a distributed system. The

current value of the clock can be inspected by a user process. The clock also serves as the mechanism by which timestamps can be affixed to messages sent between sites.

2. Every site in the network is at any time in one of two states, UP or DOWN. The UP state is characterized by correct operation and by timely response to messages sent by other sites; a site in the DOWN state is not operating at all. Transitions between these states (called "crashes" and "recoveries") occur instantaneously with respect to the global clock. A user process is provided with the ability to ascertain the current status of any site in the network, and to request that it be informed when that site changes its state.

3. The RelNet provides a reliable communication service that makes two guarantees. First, that messages sent from one site to another are received in the same order that they are sent. Second, that, on user request, a message can be marked for guaranteed delivery. That is, a message can be sent to a site that is DOWN, and the RelNet will guarantee that the message will be received by that

site upon its recovery, even if the sending site is DOWN at the time.

4. A facility for distributed transaction control is provided. This provides for a process running at one site to coordinate the activities of a number of "cohort" processes that are running at different sites and seeking to realize a global activity. The principal feature of this facility is its global abort/commit capability, which enables the controlling process to instantaneously cancel the transaction at any point or to signal its successful completion and cause the results to uniformly take effect at all involved sites.

3.1.3 Layered Architecture

The RelNet is itself organized and implemented as a series of software layers, each of which provides a subset of the facilities as a whole; furthermore, the lower layer capabilities are utilized in implementing those of the higher levels. The lowest level of the RelNet is known as the Virtual Network, and provides the global clock and site status features described above. The next layer is Guaranteed Delivery, which enables a user to send messages to down sites, with the assurance they will be received when the site recovers. The topmost layer of the RelNet is the Transaction Control layer, which provides the ability to manage and coordinate a distributed transaction and deal with it atomically. We believe that this layered architecture contributes to the comprehensibility and implementability of the system as a whole.

3.1.4 Catastrophes

The RelNet is a system constructed out of discrete components: sites and communications lines, each of which is subject to failure. It is our goal to achieve the desired level of reliable functionality by techniques that will allow for failures of some number of these components. In other words, the RelNet is designed to be resilient to the failure of some of its parts, and to function correctly so long as enough of the components behave correctly. However, no multi-component system can survive the failure of too large a number of its constituents, and the same is true of the RelNet. When "too many" failures occur, the result is termed a catastrophe. Under catastrophe situations, the RelNet is not guaranteed to operate correctly. In some cases, it will simply fail to provide one of its functions, while in others it may operate in unanticipated and unpredictable ways. Although some catastrophe situations can be automatically detected by the RelNet, others can only be observed from outside the system. In either case, manual intervention by a system administrator or other responsible human authority is necessary in order to

rectify the situation. This will usually entail shutting down part of the system and reinitializing it. The various catastrophes that can befall the RelNet facilities are described together with the individual mechanisms that realize them.

It is a principle of the RelNet design that, although catastrophes can not be entirely avoided, they can be made arbitrarily unlikely by the increased replication of reliability mechanisms. In other words, the reliability of the RelNet is parameterizable in terms of such factors as the number of sites preforming various backup functions. The price to be paid for such increased reliability is of course commensurate increased overhead. The tradeoff between the two is one that must be made by the system administrator, depending on such factors as the requirements of his application and the individual reliabilities of his system components. Having made his decision, the system administrator implements it by selecting values for a number of parameters that are identified in the paper.

3.1.5 Assumptions

The components out of which the RelNet is constructed are computer sites and communications lines connecting them. We assume that the communications lines are already organized into a basic computer communication network, with a conventional set of capabilities. Consequently, failures of individual communications lines are not explicitly considered or addressed by the RelNet; they are the province of the underlying network. That is, the RelNet assumes that the underlying network will detect the failure of a communication line between two sites and employ others to send messages between them. Furthermore, the RelNet assumes that the network at all times remains fully connected; that is, that there is at all times some path of communications lines between any two sites in the system. Should this assumption be violated by the failure of key communications lines that result in certain sites being disconnected from others, the result is a RelNet catastrophe, known as a partition catastrophe. Techniques for detecting and coping with such a situation are dealt with in the Appendix. Other communications failures are not addressed in this paper. Further assumptions about

the capabilities of the communications network are described in a subsequent section.

We also make assumptions about the way in which the computer sites of the system may fail. First, we assume that every site is equipped with a "stable storage" mechanism. This is a device that enables any site to ensure that critical information that it has received can be stored in a way that will enable it to survive a failure of the site. Techniques for implementing stable storage are discussed in [LAMPSON and STURGIS]. We do allow for failures to occur asynchronously, and at any point in the system's operation. However, we do by and large restrict our attention to "clean" failures, in which the site completely ceases operation. We further assume that a site that is recovering from failure is aware of that fact and can be made to institute suitable recovery procedures.

3.1.6 Structure of the Section

This section seeks to set forth the basic facilities of the RelNet and a particular set of implementation mechanisms that can be used to realize them. Our presentation follows the layered architecture of the RelNet itself. For each layer, we identify the functionality that it provides and then describe its implementation. In general, the implementation of each layer will be expressed in terms of the facilities provided by the lower layers of the system. There are two aspects to the implementation of each RelNet mechanism; the first concerns how sites are to behave under normal operation, and the second how they react to the failure and recovery of sites (including themselves). Consequently, an important part of the RelNet implementation is concerned with how a recovering site manages to bring itself back into normal operation. In the RelNet, site recovery is also a layered operation, corresponding to the layers of RelNet implementation. Thus, a recovering site will first execute the lowest level of recovery mechanism, then the next, and so on, until it has completed the entire process; at that point, it is considered to be fully recovered.

Distributed system reliability is an extremely intricate problem. Herein, we focus on the fundamental concepts of the mechanisms employed in the RelNet, and indicate by footnote where they need extension in order to handle special cases. Space does not permit the full specification of all variations of the basic techniques that have been developed to address pathological situations.

3.2 The Message Transmission Layer

The lowest level of the RelNet consists of the basic message transmission facilities of the underlying computer network on top of which the RelNet is constructed. For our purposes, we shall assume that this layer provides facilities for sending and receiving messages between sites. However, no guarantees are made by this layer that a message that is sent by one site will be actually received by the intended destination. The receiver might fail (i.e., become incapable of receiving messages) before the message arrives there. The only guarantee that the Message Transmission Layer does provide is that for any sender-receiver pair, messages are received in the order in which they are sent, and that during a period in which

the receiver is up, it receives all messages sent to it by the sender between any two that it does receive. In other words, if receiver A receives two messages from sender B, then they are received in the order in which they were sent; furthermore, if A did not fail between the receipt of these two messages, then it also received any other messages sent it by B between those two. The failure to meet this guarantee is a RelNet catastrophe, which can have a variety of impacts on the higher levels of the RelNet and on SDD-1 user processes.

In general, the only way for a sender to be sure that a message has been received by the destination is to require an acknowledging response from the destination site. Such protocols, however, are not provided by the Message Transmission Layer, but are the responsibility of programs that use it. In particular, higher levels of the RelNet expect to receive various responses to or acknowledgments of the messages that they send. These responses are typically returned by the same level of the system that is responsible for issuing the original message. Frequently, the sender will employ a time-out mechanism to limit the amount of time it will wait for such a response; if no such response is received within this period, the sender will assume that the intended recipient is down and will take appropriate action. We shall see several instances of this in subsequent parts of this section.

3.3 The Global Time Layer

3.3.1 Introduction

Since it is a distributed system, SDD-1 must possess some mechanism to allow it to coordinate and synchronize actions being performed at different sites in a network. It employs a global clock mechanism for this purpose. A clock is simply a uniformly increasing counter whose values can be associated with events; this associationb is known as timestamping events. Timestamps must be consistent with the order of occurrence of events, so that a later event has a greater timestamp. Among the events that need to be consistently ordered by timestamps are the sending and receiving of messages between sites and the failure and recovery of sites. The principal function of the Global Time Layer of the RelNet is to provide SDD-1 with such a consistent and accurate global clock mechanism. In order to do so, it must also encompass the functions of message transmittal and reception and the monitoring of site status.

Specifically, the Global Time Layer presents to higher levels of the RelNet and to SDD-1 a virtual network with the following characteristics: there exists with the network a single global clock, by means of which events at any site in the system can be timestamped and thereby ordered; that every site in the network is at any time in one of two states, UP or DOWN; and that transitions between these states, called crashes and recoveries, occur instantaneously with respect to the global clock. The interface to the Global Time Layer provides higher levels of the RelNet and user processes the following abilities: to send a message to another site, which will be timestamped with the value of the "global clock" at the time that it is sent; to receive a message; to determine the value of the Network Clock; to determine the current status of any site in the network; and to request that a "Watch" be placed on any site, so that the requesting process be notified when that site changes its status.

The central concept in the Global Time Layer is that of a global clock; this is a mechanism used to achieve an ordering of events occurring in a distributed system. [LAMPORT] observes that events in a distributed system should be considered to be partially ordered rather than totally ordered. That is, a relative order need be established between events in different processes only if

there is some communication between the processes that could serve to pass information about event occurrence from one to the other and thus enable knowledge of one event to influence the other. A global clock can be used to order events, by associating with each event the value of the clock at the time of its occurrence. The clock value associated with an event is known as its timestamp. The absolute value of a timestamp is of no interest; only the relative values of those of ordered events need concern us. (In other words, the principal requirement of a global clock is that it support and model our notion of causality.) The following rules determine when two events need have a defined order:

1. Events within a single process are totally ordered by their execution sequence.

2. If process B learns of event1 in process A before performing event2, then event1 must precede event2.

In the sequel, we shall see how the Global Time Layer is implemented by means of local facilities (especially local clocks and status tables). The design task with which we are faced is to coordinate local clocks and local status table in such a way as to present an interface that will simplify the design of procedures operating outside the Global Time Layer.

The implementation of the Global Time Layer is itself a layered one; however, the lower layers do not necessarily provide coherent and useful packages of capabilities, but are designed so as to realized a structured implementation of the Global Time Layer. Each layer in the implementation performs its functions by calling upon the facilities provided by lower levels. In some cases, similar facilities are provided at several levels; for example, each layer has Send and Receive primitives. However, the clean separation of the layers should contribute to the system's understandability and implementability.

3.3.2 The Local Clock Layer

The first level of the Global Time Layer is the Local Clock Layer. This implements a local clock facility that is used by higher levels of the system to simulate a uniform and consistent network-wide global clock. A local clock is simply a monotonically increasing counter that is logically independent of any real time measurement. The local clock can be read to provide timestamps for events occurring at the site. In particular, a timestamp will be assigned to every message sent from one site to another.

These timestamps are constructed by appending the current local clock value (as high-order bits) to the unique site identifier (as low-order bits). After a timestamp has been requested, the clock value will be incremented, so that the next timestamp assigned will differ from the last one.

In brief, the interface to the Local Clock Layer consists of the following functions:

- Readclock(), which returns the current value of the local clock.
- Bumpclock(n), which increments the value of the clock to be greater than the value n.
- Sendmsg(m,d), which assigns a timestamp to the message m and dispatches it to its destination d.
- Receive(), which receives a (timestamped) message.

In order to support the global ordering of events demanded by a global clock, the Local Clock Layer will examine the timestamp of each message that it receives. If its value is greater than that of the current local clock, then the Local Clock Layer will bump the local clock beyond the timestamp value. In this way, the (local) time at which a recipient receives a message is greater than the (local) time at which the sender sent it. (This capability is

necessary but not sufficient for implementing a full global clock; the remaining issues are dealt with in a subsequent section.)

During site recovery from failure, the Local Clock Layer simply sets the value of the local clock to be 0. Subsequent receipt of timestamped messages from other sites and attendant clock manipulations will restore the local clock to an appropriate value.

In addition to the logical clock facilities just described, the RelNet depends on the existence of a local real-time clock at each site. This clock is principally used to implement a time-out feature, by means of which a site that does not respond to a message within a given period is assumed to be no longer operating. In general, there need be no relationship between a site's real-time and logical clocks; the former is used to measure actual time, while the latter is for timestamping events. However, in some cases (described in a subsequent section), it is desirable to ensure that over a period of elapsed real-time, an equal amount of logical time will also have elapsed. To this end, the two clocks are kept in rough synchrony by the following technique. The two clocks are commensurable, in that they both employ the same units. The real-time clock is coarse-grained, the

logical clock fine-grained. E.g., each time a new timestamp is required, the value of the logical clock will be incremented by 1, while each half-second (say), the real-time clock will be incremented by 10,000 (say). In addition, each time the real-time clock is updated, it has the same effect on the logical clock as does the arrival of a timestamped message from another site. That is, if the new value of the real-time clock is greater than that of the logical clock, the latter is pushed beyond that value. The value by which the real-time clock is incremented is chosen to be large enough so that as a rule, the logical clock will not advance that much on its own within the interval between real-time clock advances. (Occasional lapses from this assumption are not critical.) The net effect of this mechanism is to keep the logical and the real-time clocks a bounded amount apart. In this way, elapsed real-time can be roughly measured on the logical clock when necessary. 1

This rough synchrony between logical and real clocks has two additional desirable consequence. First, it keeps the logical clocks at different sites approximately synchronized (assuming that their real-time clocks are).

---

1 It should be noted that the basic mechanism presented above is vulnerable if some site has a "runaway" logical or real-time clock. Additional mechanism can be used to resolve this problem.

This is desirable for reasons of efficiency in the SDD-1 concurrency control mechanisms [BERNSTEIN et al b]. In addition, in the event of a catastrophe that requires logical clocks to be manually reset, the system administrator can employ the real-time clock in this process.

3.3.3 The Local Status Layer

The Local Status Layer provides a set of primitives for manipulating and utilizing a site's Local Status Table, which is used to represent the site's view of the condition of all other sites in the network. These primitives are used to support the simulation of the global clock and in informing user processes of the current status of sites with which they seek to communicate.

For each site in the network (including itself) a site's Local Status Layer maintains an entry in the Local Status Table that specifies the condition of that site. (The entry for the site itself is handled in a special way, as discussed below.) Entries are made into the table upon instruction from higher layers of the Global Time Layer implementation. The status values that may be entered for a site are only UP and DOWN.

The interface to the Local Status Layer provides the following capabilities:

1. Sending and receiving messages.

2. Setting the status value of any site.

3. Requesting that a Watch be placed on a given site (or removed from it).

The Watch facility enables higher levels of the RelNet and user processes to state that they wish to be informed when a given site achieves a specified status. (Thus, there are two types of Watch; one which waits for a site to become UP and one for it to go DOWN.) The Local Status Layer will determine when that situation obtains and interrupt the requesting process to inform it that the condition has been met. (If the site is already in the designated state when the Watch is issued, the call to set the Watch will immediately return.)

The Local Status Table has an entry for each site in the network, which states whether that site has last been observed to be UP or DOWN. Furthermore, flags may be set to indicate if a Watch has been set on the site and by which user process.

Detection of site status change (either a crash from UP to DOWN or a recovery from DOWN to UP) is performed by a higher level of the Global Time Layer implementation. When such a change is detected, the Local Status Layer is instructed (by means of the MarkDown and MarkUp primitives) to change the appropriate entry in the Local Status Table. It is at this point that any processes that have requested a watch against the site will be interrupted.

When the Local Status Layer is given a message to send to a site, it first inspects the Local Status Table entry for that site. If the site is listed as DOWN, it discards the message and informs the issuing process that the message could not be delivered because the destination site is down. (The sending process could then decide to take other steps, or it could issue a recovery watch on the site in question and resend the message when that event occurs.) If the site is listed as UP, the Local Status Layer sends the message to the destination, using the send primitives of the Local Clock Layer.

All messages destined for user processes or higher levels of the RelNet also pass through the Local Status Layer. Upon receiving a message, the Local Status Layer will first check the status of the sending site. (Let us call

the sending site A and the receiving site B.) If the site A is marked as UP at B, then the message is simply passed through. If the site is marked DOWN, then the nature of the message is examined; one message type is handled differently from all others. An I'M-UP message is sent out by a site upon its recovery; such messages are sent and processed by a higher level of the RelNet. When B's Local Status Layer receives an I'M-UP message from A (a site that it has marked as DOWN), it passes through the message to be processed by a higher level of the system. If B's Local Status Layer receives any other kind of message from A, then B had incorrectly assumed that A was down (when in fact it had probably only been slow to respond to some earlier message). However, B may already have taken some action based on the assumption that A was down; consequently, it must force A to indeed be down. This is accomplished by sending A a YOU'RE-DOWN message. This message, when processed by A's Local Status Layer, will cause it to behave as though it had indeed failed, thereby validating the assumption made by B. (Subsequent recovery of A will then be instituted.)

In addition to issuing YOU'RE-DOWN messages, the Local Status Layer also has responsibility for processing incoming YOU'RE-DOWN messages. Upon receiving such a message, it should cause the local site to crash (and then

recover). (This might be accomplished simply by transferring control to the RelNet recovery module.) This should be done even if the sender of the YOU'RE-DOWN message is a site that itself is marked as DOWN. Such an occurrence indicates that two sites had made similar and mistaken assumptions about each other. In this case, the receiving site both issues a YOU'RE-DOWN message to the sender, and acts on the YOU'RE-DOWN that it received.

On recovery, the Local Status Layer simply sets the status of all sites in the network to be UP, except for its own status; that is set to be DOWN. Subsequent analyses of the responses to the I'M-UP messages (which are received and processed by a higher level of the system during recovery) will cause these values to be more accurately set.

3.3.4 The Global Clock Layer

The purpose of the Global Clock Layer is to present to higher levels of the RelNet and to user processes the view that the network as a whole possesses a single, uniform clock, which is used by all sites to timestamp messages and thereby to order events. That is, above this level it will appear that timestamps are assigned by a single global clock residing within the RelNet. The essential property of a global clock is that it be consistent with inter-site event ordering. This means that if an event occurs at site A at time t1, then an event that occurs at site B after B learns of the A event must occur at time t2, where t2 > t1. In reality of course, there is no "global clock"; rather, each site operates according to the timestamps issued by its own local clock. Therefore, the goal of the Global Clock Layer is to simulate a global clock by means of a collection of independently running local clocks.

The Global Clock Layer's interface provides the ability to send and receive messages; to examine and increment the value of the global clock; and to assure (by explicitly

crashing it) that a site assumed to be DOWN is indeed in that state. In addition, the primitives of the Local Status Layer for manipulating the Local Status Table are passed through to higher levels of the RelNet.

The local clock mechanism described above almost realizes a global clock. That is, by timestamping every message and incrementing the local clock upon receipt of a message to be greater than the timestamp on the message, the collection of local clocks almost provides an ordering of events occurring at different sites that models their actual times of occurrence and influence on one another. However, there is a way in which the local clock facility does not truly achieve a network clock; this failure is caused by implicit communication through the observation of a site failure. In our discussion of local clocks, we assumed that all communication between sites occurred through the medium of timestamped messages, and that consequently incrementing a local clock beyond the timestamp of any incoming messages sufficed to appropriately order events in the system. However, the detection of one site's failure by another also represents a form of inter-site communication, and it too must conform to the requirements of a global clock; that is, the (B-local) time at which B discovers that A has crashed must be greater than the (A-local) time at which A did

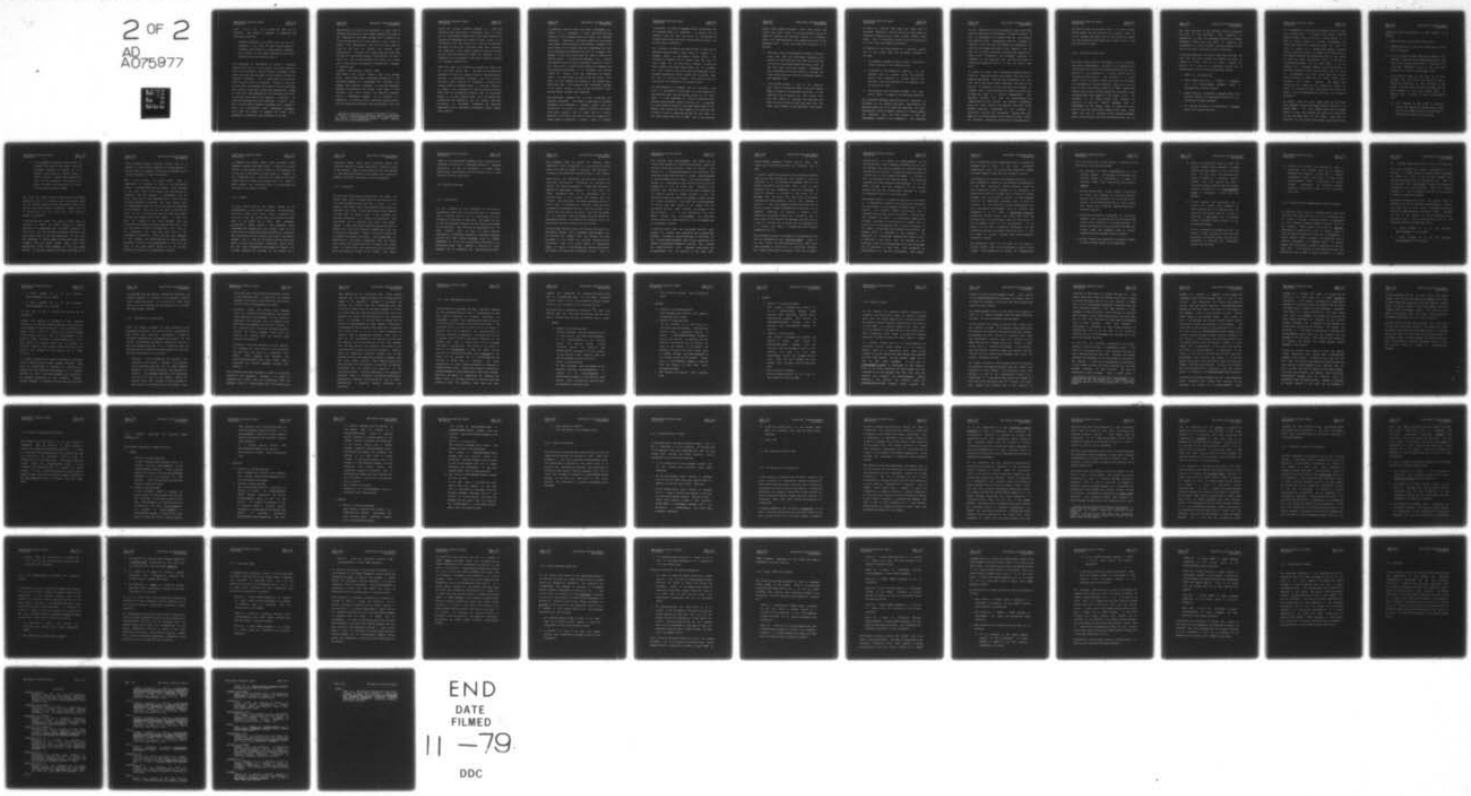
crash. In other works, the detection of a site's crash is equivalent to the receipt of a (hypothetical) I-HAVE-CRASHED message, which is however untimestamped. When B detects A's failure, B should increment its local clock to be greater than the value of A's local clock at the time A failed. Unfortunately, it requires additional mechanism to enable B to determine the (A-local) time at which A failed, since A's clock is unavailable for inspection after A has crashed.

The Global Clock Layer uses the mechanisms of guardians and Timesignal messages to construct a true global clock out of a collection of local clocks. The basic concept of this mechanism is that for each site, a set of "guardians" is maintained, each of whose clocks is guaranteed to be at most a constant value less than the value of the guarded site's clock; this is achieved by means of special messages (Timesignals) sent among these sites. Then when the failure of site A is detected by site B, B will be informed by one of A's guardians of an upper bound on A's clock at the time it failed; B can then bump its own clock to be greater than this value. This will achieve the event ordering demanded by a global clock.

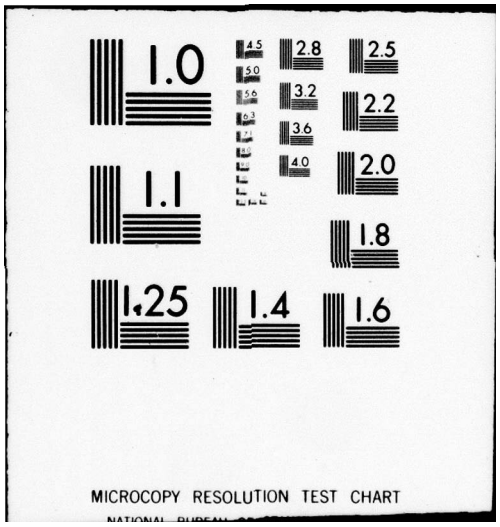
First we shall describe how guardians are implemented. Each site W has associated with it a fixed set of guardian

AD-A075 977 COMPUTER CORP OF AMERICA CAMBRIDGE MA F/G 5/2
A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND AND CONTRO--ETC(U)
JUL 79 N00039-77-C-0074
UNCLASSIFIED CCA-79-23 NL

2 OF 2
AD
A075977



END
DATE
FILMED
11 -79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART

sites, G1,G2,...,Gk. W is called the ward of its guardians. The system is designed to guarantee the following constraint:

- Guardian Constraint: If a site W and one of its guardians G are both UP, then Clock(G)+Tdelta > Clock(W), where Tdelta is a fixed system parameter and Clock(x) is the latest timestamp to have been issued by the local clock at site x.

This guarantee is implemented by having W maintain counters that tell it a lower bound on the clock values of each of its guardians. Before issuing a timestamp, W must check that this new timestamp can not possibly violate the guardian constraint. If it might cause its violation, then W can not issue the timestamp; instead, it will have to wait until new messages are received from the guardians that inform it that their clocks have advanced far enough so that the new timestamp will be consistent with their clocks under the terms of the guardian constraint. To avoid ever getting into such situations, in which it will have to wait before issuing a timestamp that it wishes to use, W will periodically issue "Timesignal" messages, which have the effect of keeping the clocks of W's guardians in relatively close proximity to W's own.

Specifically, for each of its guardians G, each site W maintains LBClock(G,W), which simply contains the value of the timestamp on the latest message from G received by W. (W can be sure that if G is up, then G's local clock is greater than LBClock(G,W), and if it is down, then G's clock at the time of failure was greater than LBClock(G,W).) Our goal is to ensure that W's clock does not get more than Tdelta ahead of LBClock(G,W), for each G that is up. This is accomplished by having the Global Clock Layer monitor the values of LBClock(G,W). Whenever it observes that

Clock(W) > LBClock(G,W) + Tdelta - RTMD,

where RTMD is the typical network round trip message delay, then W should send a Timesignal message to G. The Timesignal message is a timestamped but otherwise null message that requires a response. (A received Timesignal message is processed by the Global Clock Layer of the recipient site, which returns the expected response, itself a timestamped message. If no response to a Timesignal is received within a specified Timeout period, then the issuing site assumes that the guardian site has

---

1 Additional mechanism is required to address a situation in which a ward mistakenly believes that a guardian site has failed. This mechanism centers around special handling of YOU'RE-DOWN messages.

failed and invokes Crashsite against it. 1 When the condition just cited holds, it indicates that G's clock may be approaching being more than Tdelta behind W's; sending the Timesignal will cause G's local clock to be incremented past the value of W's clock at the time that the signal is sent, and bring them closer into proximity. The particular value RTMD is chosen so that by the time W's own clock has advanced by RTMD, the response to the Timesignal can be expected to have been received, enabling W to increment LBClock(G,W).

(Note that there is an interaction between the logical and the real-time clocks here. The goal of the Timesignal messages is to keep the logical clocks of the guardian and the ward in approximate synchrony, yet the condition governing their issuance is expressed in terms of the average round trip message delay, a real-time quantity. In general, the elapsing of a real-time period (such as RTMD) would not necessarily coincide with an equivalent change in a site's logical clock. It is to address this problem that the coupling of the two clocks, described previously, is performed. Therefore, the specified computation can be performed exclusively using logical clock values.)

To summarize, before a site W can issue a timestamp (i.e., assign it to a message), it must make sure that the issuance of this timestamp does not violate the Guardian Constraint. Therefore, the Send primitive of the Global Clock Layer, before passing on an outbound message to the lower layers of the Global Time Layer implementation, must first determine the timestamp that will be assigned that message by the Local Clock Layer and check it against the LBClock values. If issuing the timestamp (i.e., sending the message) would violate the Guardian Constraint, then W must wait until it receives some additional messages from its guardians that enable it to increment the LBClock values and thereby allow the timestamp to be legally issued. (In other words, in such a case the Global Clock Layer will not complete the sending of the message until that later time.) The purpose of the Timesignal message is to avoid forcing a site to wait (arbitrarily long) periods before issuing a new timestamp.

Some additional comments are in order concerning the guardian mechanism. The number of guardians is a system parameter, and represents a tradeoff between cost and reliability. In order to operate correctly, the Global Clock Layer requires that at least one of a site's guardians be UP while the site is down; this argues for a larger number of guardians. However, there is expense

(principally in terms of message traffic) associated with an increased number of guardians. It should also be observed that one site can serve as the guardian of several other sites, and that in particular two sites can be each other's guardians.

The Crashsite procedure has been alluded to above as the mechanism employed when a site fails to respond to a message within an anticipated timeout period. The functions of Crashsite are to ensure that the site is really DOWN, and not just slow to respond; to mark the site as DOWN in the Local Status Table; and to increment the local clock in such a way that the timestamps of any messages subsequently locally issued will be greater than the time of the site's failure. (This latter action is required for the simulation of a global clock.)

The implementation of Crashsite has two versions: the first is performed by a site that is a valid guardian of the timed-out site, and the second is used in all other cases. A valid guardian of a site is defined to be a guardian that believes itself to be up; i.e., a guardian of a site whose own value in its Local Status Table is UP. (Recall that during the first stages of its recovery from a failure, a site is operating but has its own entry in its Local Status Table set to DOWN. This is not switched

until a later stage of recovery. Until that point, the site's own local clock is not yet accurate, and so it ought not perform the guardian version of Crashsite, since, as described below, that impacts the clock of the timed-out site.) In the first case, the procedure is as follows:

1. The local clock is incremented by Tdelta. Since the local site (the one performing Crashsite) is a guardian of the timed-out site, the local clock can not be more than Tdelta behind the ward's clock at the time that the latter crashed. Consequently, this incrementation has the effect of pushing the local clock value past the last timestamp issued by the timed-out site.

2. A YOU'RE-DOWN message is sent to the timed-out site, and it is marked as DOWN in the Local Status Table (by means of the MarkDown primitive). If the timed-out site is actually UP, this action will have the effect of crashing the timed-out site while its local clock is still less than than the value just assigned to the guardian's local clock.

It should be observed that these two Steps must be performed atomically; that is, the local clock cannot be accessed or updated by any other process between these Steps. This is necessary to ensure that site status and the clock value are mutually consistent.

If Crashsite is not being invoked at a guardian against one of its wards, then the following procedure is followed:

1. A CrashReport message is sent to some (arbitrarily chosen) guardian of the timed-out site.

2. A response to this message is received. (If the guardian does not respond within the time-out period, then Crashsite is invoked against the guardian, and a new guardian for the original site is selected for Step 1.)

3. The timed-out site is marked as DOWN in the Local Status Table, by means of the MarkDown primitive.

The CrashReport message, when received by a guardian, is processed by its Global Clock Layer. Specifically, the guardian behaves as if it itself had timed-out the ward. The guardian invokes Crashsite (version 1) locally against the timed-out site and then returns a (null but timestamped) response to the CrashReport. The timestamp

on the response will be the guardian's local clock value after it completes local execution of Crashsite (which in turn is guaranteed to be greater than the lock value of the timed-out site). Consequently, when the response is received by the the site that issued the CrashReport, its clock will be pushed past that of the timed-out site at time of its crash, consistent with the order imposed by a global clock. Furthermore, the secondary execution of Crashsite at the guardian will have had the effect of crashing the timed-out site if it had not really been DOWN.

It should be noted that performing Crashsite does not entail no'ifying all sites in the network that the site in question has failed. Only the site discovering the fact (and one of the failed site's guardians) need know of the failure. Our approach is that only sites that attempt to interact with a site need know its status, and that each of these can learn of a failure independently. This approach obviates the need for synchronizing the communication of site failure and recovery information among all other sites in the network; this latter issue becomes especially troublesome in the presence of additional failures and recoveries. Instead, each site makes its own determination of the status of other sites. The necessary consistency among these interpretations is

provided by means of the global clock. If a site recovers before another one has learned of its failure, then the I'M-UP message (see below) and its processing will assure that any assumptions made about the failed site's clock are universally upheld.

3.3.5 The Global Status Layer

The Global Status Layer is the topmost level of the Global Time Layer implementation. As such, it is responsible for coordinating the various facilities provided by the layers beneath it and presenting a virtual network with the following characteristics: the existence of a single global clock by means of which all events in the system are timestamped and thereby ordered; every site is at any time in one of two states, either UP or DOWN; transitions between these states, called crashes and recoveries, occur instantaneously with respect to the global clock. The interface to the Global Status Layer provides the abilities to send and receive messages that are timestamped by the global clock, to inquire as to the state of any site, and to set a Watch on any site. These latter two sets of routines return <status,timestamp> pairs, which indicate that when the Network Clock was at

the time indicated by the timestamp, then the status of the site was that specified. This capability is implemented so that the timestamp returned is a current timestamp, rather than one which is obsolete and consequently of little interest, and are demanded in this form by the SDD-1 concurrency control mechanisms.

As has been mentioned, the Global Time Layer provides the view that any site is either UP or DOWN at any time. In reality, of course, this is a fiction. Rather, the behavior of a site in a distributed system can be characterized by one of the following four states:

1. DOWN: i.e., not operating.

2. SLOW: Running, but slow to respond to messages; i.e., not acknowledging messages within a pre-specified time-out interval.

3. FAULTY: Running, responding to messages within the time-out interval, but operating incorrectly; i.e., producting erroneous messages.

4. UP: Running correctly, and responding to messages within the time-out interval.

It is impossible to accurately distinguish among all these four possibilities. A foreign site cannot determine whether another site is failing to respond because it is DOWN or merely because it is SLOW. Consequently, the Global Time Layer merges these two cases. Any site that fails to respond to a message that demands a response within a time-out interval is assumed to be DOWN; however, the system recognizes the possibility that it may merely be SLOW and takes appropriate action (in the form of YOU'RE-DOWN messages). On the other hand, it is not possible for a message handler, such as the RelNet, to distinguish between a site's being FAULTY and its being UP. Consequently, the Local Status Layer only records sites as being UP or DOWN, and the Global Status Layer maintains this views. (However, the RelNet does incorporate the capability for components outside the RelNet to explicitly crash a site that is believed to be FAULTY. This is accomplished by exporting the Crashsite procedure.)

We observe that the Local Status Table and the Global Clock together realize the property that if the site A is marked DOWN i B's Local Status Table, then A crashed before its local clock reached the value of B's clock when it finds the DOWN entry in the table. This fact is exploited by the routine that handles inquiries into the

status of a site. The operation of this routine is as follows:

1. Read the value of the Global Clock into t.

2. Examine the entry in the Local Status Table for the site in question.

3. If it is listed as DOWN, then return <DOWN,t>; this indicates that the site crashed before time t (and that it will recover after time t). Note that t is a "current timestamp", since it has just been returned by a read of of the global clock.

4. If the site is listed as UP, then it is still possible that the site is not really UP, that it failed at a time prior to that returned by the global clock and that this fact has simply not yet been recorded in the Local Status Table. To clarify this, a Probe is sent to the specified site.

   a. If a response to the Probe is received within the specified time-out period, then the site is indeed known to be up at time t, and the pair <UP,t> can be returned.

b. If no response is received, then it must be presumed that the site has failed. Then the Crashsite procedure is called, as it is whenever a site fails to respond within a timeout period. Control then transfers to step 1 of this procedure. The purpose of so doing is to get a new clock value at which to state that the site is DOWN.

The first two steps of this procedure must be performed atomically; that is, no changes to the Local Status Table can be allowed between the time the Global Clock is read and the Local Status Table is inspected. (Again, this is to ensure that the site status and clock value are mutually consistent.)

A user process may request the Global Status Layer to institute a Watch (of either failure or recovery varieties) on a designated site. The Watch primitive at this level will call the Watch primitive provided by the Local Status Layer, to set the table entry. If the user has requested a failure watch (notification when a site crashes), then the Global Status Layer will then periodically issue Probe messages to the site being watched. If the site responds, the watch continues; if it

fails to respond within a time-out period, then it is assumed to have failed, Crashsite is invoked against it, and the caller is informed of the failure (together with a relevant timestamp, as specified above).

When a site recovers, the Global Status Layer is responsible for bringing it back to full operational status. It accomplishes this by sending I'M-UP messages to all other sites in the network, and then waiting for responses. Each such response is timestamped with the local clock value at the responding site. (Each time one of these responses is received, the Local Status Layer will automatically push the local clock past the timestamp on the response. This will have the effect of pushing the clock of the recovering site past any time that another site may have thought that it was DOWN.) If some site does not respond to the I'M-UP message, then the recovering site invokes Crashsite against that site; this will also have the effect of pushing the local clock of the recovering site past the clock value of the (presumably) failed site. After each site has responded or been crashed, the recovering site is ready to resume operation; it does so by calling MarkUp to set its own value in the Local Status Table to be UP and then transferring control to an appropriate location.

By symmetry, the Global Status Layer processes I'M-UP messages received from other sites. It does so by calling MarkUp on the issuing site to cause it to be marked as UP in the Local Status Table; it then issues a response to the recovering site. Note that the change to the Local Status Table will cause any processes that had issued a Watch against the recovering site to be interrupted and notified of its change in status.

3.3.6 Summary

The Global Status level is the topmost stratum in the implementation of the Global Time Layer. Its primitives, plus some of those of the lower levels, constitute the facilities that the Global Time Layer provides to other parts of the RelNet and to SDD-1 user processes. Specifically, these include a Send primitive, which timestamps each message with the current value of the Global Clock; an associated Receive primitive; a primitive that returns the current status of any site in the network together with a value of the Global Clock at which that status is valid; and the Watch facilities, which notify the user when a designated site changes its status. All of these features are provided in the context of a

consistent Global Clock, which accurately models the relative ordering of events occurring at different sites in the network. Some of these facilities will be employed in the remaining sections of this paper, while others are needed by the SDD-1 concurrency control mechanisms.

3.3.7 Catastrophe

The principal catastrophe situation that can befall the Global Time Layer is brought about by the failure of all of a site's guardians while the site is down. If this results, then other sites in the network will be unable to ascertain the value of the failed site's local clock at the time that it failed; this in turn prevents the accurate simulation of a global clock mechanism. This catastrophe can be detected from within the RelNet, because a site performing Crashsite against the site in question will find itself unable to communicate with any of that site's guardians. However, it would be unsafe for the system to proceed in the face of this catastrophe. By neglecting to accurately synchronize with the failed site's clock, the system would fail to uphold the global clock; in other words, the clock would fail to correctly model the sequence of events in the system. The result

might be an inconsistent database, whose contents do not represent the result of a legitimate sequence of database operations. In such an eventuality, a human system administrator must manually set the clocks so as to avoid conflict with the failed site.

3.4 Guaranteed Delivery

3.4.1 Introduction

It was observed in the discussion of the Message Transmission Layer that its facilities make no guarantee that a message sent by it to a destination site will eventually be delivered to that site; the reason for this is that the intended recipient may fail before the message can be delivered. This situation is unacceptable for the SDD-1 context, since in order to insure database consistency, a transaction updating a distributed database must be sure that certain messages (such as UPDATEs [BERNSTEIN et al b]) will be eventually delivered to all destination sites. SDD-1 demands a facility by which messages may be designated for "guaranteed delivery".

Such messages would be assured of reaching their destination site irrespective of the current or future status of either the sender or receiver. This facility is provided by the Guaranteed Delivery Layer of the RelNet.

Specifically, the Guaranteed Delivery Layer affords the following functionality. Primitives are provided for sending and receiving messages. A sender may designate a message for "guaranteed delivery". In this case, the RelNet guarantees that, if the destination site is currently down, it will receive the message upon its recovery. More precisely, the RelNet guarantees that if a sender sends two messages to a receiver, where the first is marked for guaranteed delivery, then the receiver will receive the first before the second. Note that the RelNet can not guarantee that any message (even one marked for guaranteed delivery) is certain to be received, since the destination may never recover from its failure.

The Guaranteed Delivery Layer will provide the sender of a guaranteed message with a subsequent acknowledgment (via an interrupt, for example) when the message has been processed by the RelNet for eventual delivery to the destination. (This will entail making an appropriate number of copies of the message and storing them within the RelNet; this mechanism is detailed below.) Once it

has received this acknowledgment, the sender can be certain the message will reach the destination, should the destination eventually recover from its failure. Upon receipt of a guaranteed message by the destination, the receiving process must provide a further acknowledgement to the Guaranteed Delivery Layer. This acknowledgement of receipt is a guarantee by the receiving process that the message has or will be acted upon. (Typically this requires that the message has already been processed and its effects secured on stable storage or that the receiving process has placed the message on stable storage for future processing.) Only after this acknowledgement of receipt has been issued to the RelNet will the Guaranteed Delivery Layer consider the message to have been delivered. I.e., if the destination site fails before issuing this acknowledgment, the Guaranteed Delivery Layer will again provide it with the message upon its subsequent recovery.

It should be noted that the Guaranteed Delivery Layer accepts for sending both guaranteed and non-guaranteed messages. Although no special handling is performed for the non-guaranteed messages, their proper sequencing with respect to guaranteed messages is assured. Thus, between any sender-receiver pair, messages (guaranteed and non-guaranteed) will be received in the order sent.

Non-guaranteed messages, however, may be lost; such losses occur during periods when the receiving site is down.

A further capability provided by the Guaranteed Delivery Layer is the Check primitive; this enables a site to determine if it has received all messages sent to it by another site before a given time. Thus a call on this primitive has two arguments, a site and a timestamp. A positive response is returned to the caller if all messages from the given site sent prior to the given timestamp have been received; that is, the response is positive if the next message received from that site is certain to have a higher timestamp than that given as an argument. Otherwise, the response is negative. This capability is employed in SDD-1 by a recovering site to ensure that it has received all messages that were sent to it while it was down, and to ensure that all messages sent by a failed site before it crashed have been received [BERNSTEIN et al b].

The guaranteed delivery of messages is accomplished by the user of a mechanism we call a Reliable Buffer. There is one such buffer for each destination site. When a user flags a message to a down site for guaranteed delivery, the RelNet establishes the message in that site's buffer;

having done so, it returns an acknowledgement to the sending process, since (assuming the RelNet does not fail) the message will now be available for retrieval when the destination site recovers. During the recovery process of a failed site, it will request the RelNet to provide it with all messages in its Reliable Buffer. The recovering recipient will then establish these messages on its own stable storage; and upon completion of its recovery process, the site will process these messages as though they appeared normally in its input queue.

The Reliable Buffer is a mechanism internal to the RelNet; it is implemented by means of the appropriate replication and coordination of each message at several sites in the network. This approach differs significantly from a technique that has been called "persistent communication" (see [ALSBERG and DAY]). In a persistent communication strategy, a message to be delivered to a crashed site is buffered only at the sending site. In such a situation, if the sender is down when the recipient site recovers, the message cannot be forwarded to the recipient. The assumption of a persistent communication scheme is that, at some point in the future, both the sending and the receiving sites will simultaneously be up and the message can be delivered at that time. This assumption is unsatisfactory for the SDD-1 environment. SDD-1 demands

that a recovering site be immediately able to retrieve all messages sent it while it was down; persistent communication does not provide this capability, because the sender might be down when the recipient recovers.

This capability to retrieve all messages sent prior to a given timestamp is needed for the efficient implementation of the SDD-1 concurrency control protocols. Under the SDD-1 concurrency control strategy, it is necessary, in synchronizing against a crashed site, to obtain all UPDATE messages sent by it before it failed. If these messages could not be immediately obtained, then it would be necessary for the concurrency control mechanisms to wait until such time as the messages could be obtained (see [BERNSTEIN et al b]) Under a "persistent delivery" strategy as outlined above, this would mean that the synchronizing site would have to wait for the recovery of the site against which it was synchronizing. In the design of the SDD-1 reliability mechanisms, we have avoided any approach that might require one site to wait for others to recover, because such recovery might be arbitrarily delayed.

The problem with which we are faced is to design a Reliable Buffer that will be available in spite of site crashes. The presentation of a design for accomplishing

this is the main topic of this section. An outline of the remainder of this section follows:

1. We introduce the basic implementation of the Reliable Buffer. For purposes of robustness, the Reliable Buffer is replicated at a number of different sites, each replication being called a spooler.

2. We then observe that, unless special precautions are taken, the messages as stored at different spoolers will not necessarily be in the same order. However, we then demonstrate that no harm results from having such different message orderings at different spoolers.

3. We discuss alternative strategies for recovering messages from the spoolers and for switching them cleanly from a spooling to a non-spooling mode. We present the details of the particular strategy that is used, under the assumption that no spooler crashes during the message recovery process.

4. We next consider the possibility of spooler crashes at three critical points in the algorithm.

a. First, we consider the case in which the spooler crashes while messages are being removed from the spooler by the recovering site. In this case, the recovering site simply switches to a different spooler. However, the new spooler may have a different message ordering than the original spooler. The notion of an acknowledgement vector is introduced to deal with this problem.

b. We then consider the possibility that a spooler site crashes while messages are being inserted into it by sending sites. We argue that the spooler site can recover by inserting a gap marker into its message stream to indicate a point at which it may be missing messages.

c. Third, we consider the possibility that the spooler crashes during the transition from spooling to non-spooling mode, and introduce mechanisms to eliminate the undesirable effects this may have.

5. Finally, we consider the possibility that a recovering site crashes while it is removing messages from a spooler. We show that no harmful consequences result from this, so long as the acknowledgement vector is maintained on stable storage. Furthermore, the mechanism operates correctly if spoolers as well as the recovering site crash.

3.4.2 Reliable Buffer Implemented As Multiple Spoolers

As mentioned above, there is a Reliable Buffer associated with each site, whose function it is to hold messages sent to it while it was down. A Reliable Buffer is implemented as a set of physical buffers located at several sites in the network. These buffers are known as spoolers. Associated with each site is a set of spooler sites at which the Reliable Buffer for that site is implemented. A single spooler would, in general, be inadequate since it would be susceptible to crashing itself. To protect against this occurrence, a number of spoolers are used for each site. In the sequel, we will assume that while the destination site is DOWN, at least one spooler is always

UP. A RelNet catastrophe occurs if this is not the case.

The basic strategy for spooler implementation is as follows. If a sender wishes to reliably buffer a message, the RelNet will send a copy of that message to all spoolers associated with the destination site. When all the spoolers have acknowledged receipt, the message is considered to be reliably buffered. When the recipient recovers, it issues a request to any one of its spoolers to obtain its buffered messages.

One might be tempted to say that each spooler holds an identical copy of the (logical) Reliable Buffer. However, this need not be true, since a different spooler may contain the same messages in different orders. Consider the case of two spoolers S1 and S2 associated with a destination site C, and two sending sites, A and B. The following sequence of events may occur:

1. A sends message M1 to S1 and receives acknowledgement of its receipt.

2. B sends message M2 to S2 and receives acknowledgement of its receipt.

3. A sends message M1 to S2 and receives acknowledgement of its receipt.

4. B sends message B2 to S1 and receives acknowledgement of its receipt.

In this case S1 has M1 preceding M2, while in S2, M2 precedes M1.

However, the ordering of messages at both spoolers conforms to the partial ordering imposed by the sending of the messages (as discussed earlier). Such conformity will always obtain, since if a given site sends one message before another, the first must be acknowledged by all spoolers before the other is acknowledged by any. Thus, even though the message orders may be different at different spoolers, each spooler ordering conforms to the partial order imposed by the sending and is hence "correct".

It should be noted that it would be possible to guarantee that all spoolers have the same message ordering. This would be accomplished by imposing an ordering on the spooler sites and requiring that messages to be reliably buffered be sent to and acknowledged by the spoolers in the sequence determined by that ordering. However, sending messages in sequence this way requires one round

trip message delay per spooler, whereas our technique, of sending messages in parallel to all spoolers, requires only a single such delay. In any case, as we have shown above, it is not necessary for the spoolers to all contain the same message ordering.

3.4.3 Implementation Alternatives

While the general strategy of using spoolers is not conceptually difficult, the details of the implementation may become quite intricate. Furthermore, a number of different implementation strategies are possible. We can outline three general approaches, differing in the way in which new messages destined for a recovering site are handled while the spoolers are being emptied by that site. The three approaches are as follows:

- Strategy 1: Prior to emptying the spoolers, the recovering site sends a message to all sending sites indicating that they are to cease sending messages to it, either indirectly (via spoolers) or directly. Any messages destined to the recovering site are to be held at the sender. After the spoolers have been emptied, a further message is sent from the recovering site to the sending sites

directing them to henceforth send messages directly to the recovering site. In particular, any pending messages being held at the sender may now be sent, and will then be acknowledged on receipt.

- Strategy 2: Rather than holding their messages while the spoolers are being emptied, the sending sites continue to send messages to the spoolers. Eventually the spoolers will be emptied, after which point new messages will be sent directly to the recovered site. (Since we can safely assume a receiver can receive messages faster than the senders are sending them, the spoolers will eventually be emptied.)

- Strategy 3: While the spoolers are being emptied, new messages are sent directly to the recovering site where they are kept in a temporary local buffer. After emptying the spoolers, the recovering site empties this local buffer before receiving any further messages directly from senders.

The strategies have been presented in order of increasing efficiency and complexity. Strategy 1 is simple to implement but suffers from the disadvantage that new messages cannot be acknowledged until the spoolers have

been emptied by the recovering site. Since spooler emptying may be a lengthy process, this strategy was not deemed to be acceptable. Strategy 2 overcomes this problem, but suffers from the disadvantage that, even though the crashed site has recovered, messages to be sent to it must make two "hops" through the network, one to the spoolers and another from the spoolers to the recovering site. In strategy 3, only one "hop" is needed, but the details of implementation are quite complex. Particularly troublesome are the problems raised should the recovering site crash while it is in the midst of emptying its spoolers. When it subsequently recovers, there will be both old and new messages in the spoolers; the old ones must precede, and the new ones must follow, the messages that had been placed in the temporary local buffer at the recovering site during its previous recoveries. Further, these old temporarily buffered messages must be kept separate from any new temporarily buffered messages that may arrive during recovery. After complete details had been developed for this approach, it was felt its improved performance did not justify the complexity of the resulting software. (Software simplicity is an especially important issue in developing reliability mechanisms.) Consequently, Strategy 2 was selected for use in SDD-1, representing a trade-off between efficiency and simplicity.

3.4.4 Basic Implementation Algorithm

In this section, we present the basic algorithm employed by the Guaranteed Delivery Layer in implementing its send and receive primitives for messages designated for guaranteed delivery. This implementation employs several of the facilities provided by the Global Time Layer. The algorithm below is cast in terms of a set of senders, a receiver, and a set of spoolers. (The generalization to multiple receivers is immediate.) We assume that the set of spooler sites is known to the Relnet component at the sending site. Each of these sites is one of two modes (spooling or non-spooling). The collection as a whole is said to be non-spooling if both the sender and the receiver are in non-spooling mode, and is spooling if the sender and the spoolers are in spooling mode and the receiver is either down or spooling. All other combinations of local states correspond to transient or illegal global states. Global state change is effected by sending messages to cause appropriate local state changes. Basically, the sender is responsible for changing the global state to non-spooling when it discovers that the receiver is down. The spoolers, after they have been

emptied, are responsible for changing the global state back to non-spooling mode. In the basic algorithm outlined below, we do not provide for spooler crashes. These will be considered in the following section.

The algorithm is expressed by describing how each site behaves when its local mode is spooling, when its local mode is non-spooling, and upon its recovery from a crash.

1. SENDER

   a. Sender in non-spooling mode:

      To send a message, send the message directly to the receiver, and await acknowledgement. If the acknowledgement is not received within the time-out period: invoke Crashsite against the receiver; send START-SPOOLING messages to all the spoolers for the receiver; enter spooling mode and send the message as specified below.

   b. Sender in spooling mode:

      To send a message: send the message to all spoolers, and await acknowledgements. If, instead of an acknowledgement, a STOP-SPOOLING message is received from some spooler, enter non-spooling mode and send the message as specified there.

c. Upon recovery of sender: enter non-spooling mode.

2. RECEIVER

a. Receiver in non-spooling mode:
Await messages sent directly from senders, acknowledge upon receipt.

b. Receiver in spooling mode:
(If the receiver is in this mode, it is recovering spooled messages.) Upon entering spooling mode, a particular spooler is chosen. A NEXT-MESSAGE-PLEASE message is sent to this spooler. In response, the next message in the spooler will be returned (so long as the spooler is not empty); the message is stored on the site's input queue on stable storage, and acknowledgement of its receipt is sent to the spooler. Repeat until a STOP-SPOOLING message is received from the spooler; at that time, enter non-spooling mode.

c. Upon recovery of receiver: enter spooling mode.

3. SPOOLER

a. Spooler in non-spooling mode:

Upon receipt of message from a sender: if it is a START-SPOOLING message, enter spooling mode; otherwise, respond with a STOP-SPOOLING message. Upon receipt of NEXT-MESSAGE-PLEASE message from the receiver: send STOP-SPOOLING message to receiver.

b. Spooler in spooling mode:

Upon receipt of message from a sender: add message to buffer. Upon receipt of NEXT-MESSAGE-PLEASE message from the receiver: send next message in buffer to the receiver, and after it has been acknowledged, delete the message from the queue. When the last message has been deleted from the queue, enter non-spooling mode.

c. Upon recovery of Spooler:

We have assumed spoolers do not crash in this version of the algorithm.

3.4.5 Spooler Crashes

We will examine the problem of spooler crashing in two contexts. The first case is that of spoolers crashing while they are being emptied by a recovering receiver. The second is that of spoolers crashing while the receiver is still DOWN. (If a spooler crashes at any other time, it can be handled in the same way as in this second case.) After considering these issues we present the complete algorithm including provision for these spooler crashes.

If a spooler crashes while it is being emptied, the recovering receiver should switch to a new spooler. However, there is a complication here: many of the messages in the new spooler may have already been received from the former spooler. The new spooler ought not to have to send these messages. To this end, an acknowledgement vector is maintained by the receiver. This is an array indicating, for every sender site, the timestamp of the last message from that site that the receiver has received and acknowledgement. Before emptying a new spooler, the receiver sends an ACKNOWLEDGEMENT-VECTOR message, which contains the

receiver's current acknowledgement vector. Upon receipt of the ACKNOWLEDGEMENT-VECTOR message, the spooler uses it to delete all messages in its queue that have already been received by the receiver.

The acknowledgement vector is also used by the receiver to allow it to ignore messages that have been previously received , but are nonetheless received again.

We now consider the problem of spoolers that crash while the receiver is DOWN. If the spooler remains DOWN until the receiver has recovered and emptied some other spooler, no problems can arise. However, if a spooler does crash and subsequently recovers while the receiver is still DOWN, that spooler's message queue will reflect a "gap" during which it received no messages. If the receiver, upon its recovery, chooses to empty this spooler, then the receiver will not receive those messages sent while the spooler in question was down.

One simple solution to this problem would be to disallow the receiver from emptying spoolers that had crashed and recovered in this manner. The difficulty with this approach is that it will unnecessarily result in a catastrophe in a case where each spooler has crashed for some time during the period that the receiver was down; even though every message sent to the receiver is

available in some buffer, no spooler has them all. This would mean that at least one spooler would have to remain up during the entire period that the receiver is DOWN. This is an unreasonable expectation considering the fact that sites may be DOWN for very long periods of time. Instead, it would be better to have spoolers (on recovery) mark the gaps in their queues during which they were DOWN, and to have the receiver fill those gaps from messages held in other spoolers. Under this strategy, a catastrophe is prevented so long as that collection of spoolers that are up at the time of the receiver's recovery hold all of the messages that had been sent to it while the receiver was down.

To this end, the following conventions are followed: whenever a spooler recovers, it immediately places a GAP marker in its message buffer. This indicates the point at which messages may have been lost. Secondly, each sender remembers the timestamp of the last message it has sent to the receiver. 1 (This can be accomplished by maintaining at each sender an array, PMT, which contains the previous message timestamp for each receiver. PMT must be maintained on stable storage.) When a sender sends a

---

1 This message need not yet have been acknowledged. The intention is for the sender to be able to inform a recovering spooler of the last message that it may have missed.

message to a spooler, it appends to that message the timestamp of the previous message that it sent to the receiver. When unspooling, the spooler now behaves as follows. When the spooler receives a NEXT-MESSAGE-PLEASE message and the next item in the buffer is a GAP marker, a GAP message is sent to the recovering receiver. The GAP marker remains in the message buffer. Upon receipt of such a GAP message, the receiver chooses another spooler from which to obtain the remainder of its messages. (The first step in this process, as described above, is to send the new spooler an ACKNOWLEDGEMENT-VECTOR message, which it uses to delete messages already obtained elsewhere by the receiver.) The receiver then retrieves messages from this second spooler until this spooler crashes or until another GAP is encountered. At this point, the receiver will move on to another spooler; in particular, it may return to one that it had left earlier upon receiving a GAP message. (The receiver may return to an earlier spooler only after it has received at least one additional message from some other spooler. As usual, the recipient site will reestablish the interaction with the earlier spooler by sending it an ACKNOWLEDGEMENT-VECTOR message.)

In addition to deleting messages that the acknowledgement vector indicates have already been received by the recipient, the receipt of an acknowledgement vector

message by a spooler will cause it to delete any GAP markers in its buffer that are no longer operative. Intuitively, a GAP marker is operative if there may be messages for the receiver that are missing from the buffer and whose place is occupied by the GAP marker. Therefore, a GAP marker is no longer operative if, for each sender site, there is a message in the buffer following the GAP marker whose predecessor (as indicated by the previous message timestamp that is attached to each message) has already been received by the recipient. The spooler can establish if a GAP marker is operative by examining the acknowledgement vector sent it by the receiver and the messages in its own queue. After deleting the inoperative GAP markers, the spooler can resume sending spooled messages to the receiver.

We must also consider the situation in which the spooler crashes immediately after having sent a STOP-SPOOLING message to the receiver. A sender in this case may not know that the receiver has entered non-spooling mode and will continue to send messages to the spoolers that are still up (who also have not learned of the state change). To prevent this situation, the receiver, after receiving a STOP-SPOOLING message from one spooler, does not immediately enter non-spooling mode. Instead, it switches to another spooler in the usual way and attempts to

retrieve messages from it. (In most cases, the second spooler will not have any additional messages, but in the situation described above,where the first spooler crashed before notifying senders of the state change, there may indeed be some new messages there.) The receiver then enters non-spooling mode only after having received a STOP-SPOOLING message from all of the UP spoolers.

Finally, we must consider how the sender will deal with failing and recovering spoolers. The sender will crash any spooler that does not acknowledge receipt of a message sent to it; this will ensure that the spooling process is accurately begun and that messages are securely spooled. When a failed spooler recovery, the sender brings it into the spooling process by issuing it a START-SPOOLING message.

3.4.6 Crash of the Recovering Receiver

The recipient may crash while it is in the process of unspooling. Upon its recovery, it simply chooses a spooler and resumes the unspooling operation. It should be noted that the acknowledgement vector must be the same as at the time of the recipients' crash in order for messages not to be received twice. This requires that the receipt keeps its acknowledgement vector (or more precisely, the information that it contains) on stable storage. If the recipient maintains its input queue on stable storage, then the information needed to reconstruct the acknowledgement vector is available from this input queue.

3.4.7 Complete Algorithm For Reliable Buffer Implementation

The complete algorithm is summarized below:

1. SENDER

   a. Sender in non-spooling mode:
      To send a message, send message directly to receiver and await acknowledgement. If the acknowledgement times-out: invoke Crashsite against the receiver and send START-SPOOLING messages to all the spoolers for that receiver. Enter spooling mode and send the message as specified below.

   b. Sender in spooling mode:
      To send a message, append to message the current value of PMT [receiver] and then set PMT [receiver] to the current global clock time. Send message to all spoolers that are currently UP, and await acknowledgements. If, instead of acknowledgement, a STOP-SPOOLING message is received from some spooler, cancel the recovery watches against

DOWN spoolers, enter non-spooling mode, and send the message as specified there. If an acknowledgement times-out, invoke Crashsite against that spooler and establish recovery watch against it.

If a crashed spooler recovers: Send STOP-SPOOLING message to the spooler.

Upon recovery of sender: Enter non-spooling mode.

2. RECEIVER:

a. Receiver in non-spooling mode:

Await messages sent directly from senders; upon receipt, update acknowledgement-vector and acknowledge the message.

b. Receiver in spooling mode:

Upon entering spooling mode, a particular UP spooler is chosen. An ACKNOWLEDGEMENT VECTOR message containing the current acknowledgement vector is sent to the spooler. (A response is expected; if none is received within a time-out period, Crashsite is invoked against the spooler and another is selected.) Successive NEXT-MESSAGE-PLEASE messages are then sent

to retrieve messages from the spooler. If the spooler does not respond to a NEXT-MESSAGE-PLEASE within a time-out period, Crashsite is invoked against it and another spooler is selected for unspooling. If the spooler replies with a regular message, update the acknowledgement vector, acknowledge the message, and establish the message on the input queue on stable storage. If the spooler replies with a GAP message or a STOP-SPOOLING message, initiate unspooling from another spooler. The unspooling procedure terminates when a STOP-SPOOLING message has been received from all UP spoolers, at which point non-spooling mode is entered.

c. Upon recovery of receiver:

Reconstruct the acknowledgement vector if necessary; enter spooling mode.

3. SPOOLER

a. Spooler in non-spooling mode:

Upon receipt of message from a sender: if a START-SPOOLING message, acknowledge and enter spooling mode. Otherwise, respond with a STOP-SPOOLING message.

Upon receipt of NEXT-MESSAGE-PLEASE or ACKNOWLEDGEMENT-VECTOR message from the receiver: send STOP-SPOOLING message to the receiver.

Spooler in spooling mode:

Upon receipt of message from a sender: add message to buffer and acknowledge.

Upon receipt of ACKNOWLEDGEMENT-VECTOR message from the receiver: delete all previously received messages from the queue as well as GAP markers that are no longer operative and acknowledge. If the queue becomes empty, enter non-spooling mode.

b. Upon receipt of NEXT-MESSAGE-PLEASE message from the receiver:

If the next item in the buffer is a GAP marker, send a GAP message. If the next item in the buffer is a message, strip the previous message timestamp from it and forward it to the receiver, and remove the message from the buffer when the receiver has acknowledged it. If the buffer becomes empty, enter non-spooling mode.

c. Upon recovery of Spooler:

Place GAP marker in the message buffer.

3.4.8 Spooler Catastrophe

The algorithm as just described requires that at least one spooler be UP whenever the receiver is down. When this condition is not met, a spooling catastrophe may occur. This catastrophe is detected by the sender when no UP spoolers are available for spooling. The catastrophe is detected by the receiver, when all spoolers that are UP return GAP messages in response to a NEXT-MESSAGE-PLEASE message. By providing for additional or more robust spoolers, the likelihood of spooler catastrophe can be decreased.

3.4.9 Implementation of Check

As described above, the Check primitive takes a site, S, and a timestamp, TS, as its arguments. It responds TRUE if all messages from S with timestamp less than TS have already been received, and FALSE otherwise. This is implemented by the following algorithm:

1. If any message from S with timestamp greater than TS has already been delivered, return TRUE; otherwise:

2. If the input message queue contains no messages from S, and the Global Time Layer reports that S is DOWN as of time TS, return TRUE; otherwise:

3. If the message input queue contains no messages from S, issue a failure watch against S, and wait for the probe timeout interval; then go to step 1 (after this interval, either S will have been marked DOWN or a timestamped message from S (in particular, a ProbeResponse) will have been received); otherwise:

4. If the next message from S in the message input queue has timestamp less than TS, return false; otherwise:

5. Return TRUE

3.5 The Transaction Control Layer

3.5.1 The Atomicity of Transactions

In this section, we describe how the Relnet supports the atomic execution of distributed database transactions, which access and modify data items that may be stored (and replicated) at several sites in the network. Reliability mechanisms are needed to ensure the correct execution of these transactions despite asynchronous site failures (in particular, these that occur during the execution of a transaction).

Following [ESWARAN et al], we define a transaction as an atomic database operation at the user level. That is, the user is given the ability of grouping together a number of

primitive database operations and calling the result a transaction; the system must then behave as if each such transaction is processed as an atomic, indivisible, unit. A transaction is specified to the system in terms of a sequence of actions, each action being an atomic operation at the system level. Even though execution of actions from different transactions may be interleaved by the system, it must preserve the outward appearance of having executed one transaction to completion before beginning another.

The atomicity constraint guarantees, for example, that it is not possible for any transaction to read data that has been partially, but not completely, updated by another transaction. Nor is it possible for two or more transactions to interleave their read and write operations so as to result in "reader-writer" anomalies. Such an anomaly could result from a scenario in which one transaction, T1, reads a variable, x; another transaction, T2, then reads the variable x; next transaction T2 updates x; and finally T1 updates x. (Consider what happens when both T1 and T2 both increment x by 1.) The difficulty arises because T1's update is based on a data value that has become invalidated by T2's update.

It is the responsibility of the concurrency control mechanism of SDD-1 to guarantee the atomicity of transactions. Atomicity is achieved by forcing read operations to wait until appropriate update operations have completed before they are allowed to execute. (Discussion of the SDD-1 concurrency control techniques is given in [BERNSTEIN et al b] and a general survey of distributed concurrency control techniques can be found in [BERNSTEIN and GOODMAN].)

One may reasonably ask why reliability mechanisms are necessary for transaction atomicity if the concurrency control strategy is correct. There are two reasons. First, the concurrency control algorithms themselves must be made safe against site failures. Second, the execution of a transaction will typically entail the sending of update messages from one site to a number of other sites; and even with properly functioning concurrency control, the sending site may fail before having issued all of the update messages associated with some transaction. A situation in which some, but not all, of the update messages associated with a transaction have been received and processed results in database inconsistency and negates the principle of atomicity. Although the unsent messages, which are "buried" at the failed site, could presumably be issued upon the node's recovery, any read

operations waiting on the completion of that transaction would they be forced to wait until such time as the recovery actually occurred. 1 Such delays would be intolerable. It is a general principle of SDD-1 that a transaction should never be forced to wait for a site to recover in order to run to completion. 2

The first issue identified above, that of failure-proofing the concurrency control mechanism, is dealt with elsewhere ([BERNSTEIN et al b]) and will not be covered here. The solution to that problem is based on the judicious use of RelNet capabilities.

The second issue, the problem of "buried" updates, is the central focus of this section. This problem can also be conceived of as that of "reliable message broadcast", enabling a site to send a collection of messages to different sites as a single package (i.e., eliminating the possibility of the sending site failing partway through the process). The problem is resolved by an atomic commit procedure. This procedure ensures that whenever any update messages become buried, the transaction with which

---

1 Assuming that the site has not failed permanently, in which case the reading transaction would have to wait forever!

2 Except, of course, in the case where the transaction seeks to read data that is stored at the failed site and nowhere else in the network.

they are associated will be aborted; i.e. none of its updates will be performed. Only when all of the update messages for a transaction have been issued will the transaction be committed; at that point, all of the updates will be guaranteed to transpire. Thus it will not be necessary for any read operation to wait for a site to recover before proceeding. In those cases where a read might have been forced to wait for a buried update, the updating transaction will be aborted, hence obviating the need for the read to continue waiting.

It is important to distinguish the intent of our atomic commit operation from similar mechanisms proposed in other contexts. For example, the two-phase commit operation of [GRAY], upon which our atomic commit is based, is designed for systems in which an update message may be rejected (for example, as a result of concurrency control considerations). In such a scheme, any update message rejection mandates the abortion of the entire transaction. The two-phase commit protocol insures that no update takes place until all update messages have been accepted. However, this mechanism as well as other proposed variants of it (e.g. [LAMPSON and STURGIS], [REED]), may withhold the commit/abort decision until after a failed site has recovered (and given the opportunity to reject the update message). Thus, a site may hold uncommitted update

messages for long periods of time. As discussed above, this would not be acceptable in the SDD-1 environment, where other transactions may be waiting for commitment of the update messages.

3.5.2 Transaction Control Functionality

The SDD-1 mechanisms for transaction control are oriented towards a particular model of transaction structure and operation. In this model, a transaction is invoked by user command at a given site. (Each transaction is assigned a unique identifier.) The invocation creates a process at that site, which is the controlling process for the transaction. This process may then cause the creation of other (cohort) processes (at this or at other sites); the transaction will be realized by coordinated activity of the set of processes. (A process is uniquely associated with a transaction, and is identified by a (transaction identifier, process identifier) pair.) The processes associated with a transaction may communicate with one another and perform local computations. At the end of the transaction, the controlling process sends out update messages, indicating to each site the changes that are to be made to certain database elements stored at the

site. The update identifies the data elements and their new values. After this action is compelted, the controller successfully terminates (commits) the transaction, causing the updates to take effect. At any point during its execution, the controller may abort the transaction; this may be caused by the failure of a site running one of the aborts. Until the transaction is committed, it has no effects that can be seen by other transactions.

In order to support such atomic transactions, the RelNet provides the following capabilities:

1. The capability to obtain a new transaction identifier; the requesting process will be the controlling process for the transaction.

2. The capability to create a cohort process at the local or a foreign site. The new process will be associated with the same transaction as the requesting process, and will be destroyed when the transaction is committed or aborted.

3. The capability to request that a transaction be committed or aborted. Both types of request signal the completion of the transaction and may be executed only by the transaction's controlling

process. When the transaction is aborted, any updates that may have already been performed by the transaction will be undone.

3.5.3 The Implementation Environment For Transaction Control

In this section, we will examine the components from which the Transaction Control capability described above will be implemented. As the outermost software layer in the RelNet, the Transaction Control component may utilize the functionality provided by the Reliable Delivery, Status Monitoring and Global Clock components. It addition it relies on a number of facilities assumed to be provided by the local operating system or database management system at each site. These latter capabilities include:

1. The capability to create a new process and associate that process with a given transaction number.

2. The capability to delete such processes.

3. The capability to perform local database updates in a tentative mode. Updates made in this mode are not installed until they have been committed.

4. The capability to abort all tentative updates associated with a transaction, restoring the original value to updated data items.

5. The capability to commit all tentative updates associated with a transaction, causing the new data item values to be visible.

It will be noted that we are essentially assuming the existence of a local transaction control capability out of which we are building a distributed transaction control capability.

The implementation of global process control primitives in terms of message communication and the local primitives is straightforward and will not be discussed here. Our primary concern will be with the implementation of the global commit/abort facility in terms of the local commit/abort primitives. The difficult technical problem here is that of uniformly committing a transaction in the presence of local site failures and recoveries.

3.5.4 Two-Phase Commit

A two-phase commit procedure, similar to the one described in [GRAY], forms the core of our atomic commit mechanism. This procedure is described below. Let C be the controlling process of the transaction and U1, U2,... Un be those cohort processes for this transaction that perform local updating on its behalf.

Phase 1a: C issues UPDATE messages to U1 through Un. These cause the local databases to be updated in tentative mode in accordance with the instructions of the UPDATE.

Phase 1b: C waits for Guaranteed Delivery Layer acknowledgement that these UPDATE messages have been delivered or reliably buffered.

Phase 2a: C issues COMMIT messages to U1 through Un. These cause the transaction to be locally committed.

Phase 2b: C waits for Guaranteed Delivery Layer acknowledgement of these COMMIT messages.

It should be noted that, in the RelNet environment, it is not necessary for the UPDATE message to actually reach the destination process. So long as the message is stable within the Guaranteed Delivery component, it is sure to eventually reach its destination. This acknowledgement is sufficient to assure that the UPDATE will reach its destination and be executed there (in tentative mode).

This procedure is, therefore, insensitive to crashes of U1 through Un before or during its execution. It is, however, sensitive to failures of the commit process C. In particular, if C crashes during Phase 1, some or all of the updating processes will have received UPDATE messages in tentative mode for which no COMMIT will be forthcoming. In such a situation we would like to abort the transaction, and discard any UPDATE messages that have been received. More importantly, should C crash during Phase 2, then all processes (including those that have received UPDATEs but no corresponding COMMITs) should commit the transaction, to ensure uniform installation of the updates.

Our solution to this involves the use of a number of commit backup processes; these are processes that can assume responsibility for completing C's activity in the event of its failure. These backup processes are created by C before it uses any UPDATE messages; each backup, when it is created, is informed of the identity of C's cohort process. Then, if C should fail before completion the transaction, one of the backups will take control. If C failed before issuing all the UPDATEs, then the backup will abort the transaction; otherwise it will commit it. In either case, the desireds effect is realized by sending COMMIT or ABORT messages to the cohorts, as appropriate. Naturally, proper coordination among the commit backup processes is crucial; a key issue is the selection of a single commit backup process to take over in the event of C's failure. Our technique for backup selection is described in the next section. Following this discussion, we describe the atomic commit procedure incorporating backups.

3.5.5 Backup Selection Algorithm

In the event of the failure of the committing process C, we wish one, and only one, commit backup process to assume control of transaction completion. The algorithm for accomplishing this assumes an ordering of the commit backup processes, designated in order as B1, B2, ... Bm. Process Bk-1 is refered to as the predecessor of process Bk (for k>0), and process C is taken as the predecessor of process B1. Initially, each of the backup processes is watching for the failure of its predecessor; that is, Bi+1 is assumed to have issued a Watch on Bi. The following conventions are then followed:

1. If a watched backup process is found to be down, then its watching process begins to watch the predecessor of the failed process instead.

2. If process C is found to be down, the backup process that is watching it assumes control of the transaction.

3. If a backup process recovers, it ceases to be a part of the backup mechanism (i.e., it behaves as if it had stayed down).

These rules have the following consequences:

1. If C fails, at most one backup process will assume control. This will be the lowest-numbered backup that has not failed at least once during the procedure. (Note: If all backups have failed at least once since their creation until the time of C's crash, then no take over will occur. This constitutes a catastrophe and will be discussed below.)

2. If a backup process, Bc, fails while it is in control of the transaction, then again at most one backup process will take over. This will be the backup Bk that was watching Bc, at the time of its failure. Having watched Bc fail, Bk will examine each of Bc's predecessors, find them all down, and eventually discover that C itself is down. At that point, Bk assumes control.

Thus, at most one process, either C or one of its backup processes, will be in control at any given time. Having assumed control, a backup will proceed to issue COMMIT or

ABORT messages, depending on its state; this issue is addressed in the next section.

3.5.6 Atomic Commit With Backups

The following four-phase procedure is used to implement atomic commit in the RelNet. Below, C is the process initiating the commit and U1, U2,... Un are its cohort processes that perform local updating on behalf of the transaction. We first describe the procedure as executed by C.

Phase 1a: C establishes m commit backup processes B1, B2,...Bm. When it creates process Bi, C informs it of the identity of all lower-numbered backup processes and of cohort processes of the transaction.

Phase 1b: C waits for an initial message from each of the backup processes to confirm its existence. Crashsite is invoked against any backup site that fails to respond within a time-out interval.

Phase 2a: C issues UPDATE messages to U1 through Un. These will cause the local databases to be updated in tentative mode.

Phase 2b: C waits for Guaranteed Delivery acknowledgement of these UPDATE messages.

Phase 3a: C issues COMMIT messages to all B1 through Bm.

Phase 3b: C waits for each backup to acknowledge receipt of the COMMIT. Crashsite is invoked against any backup that does not acknowledge within the time-out period.

Phase 4a: C issues COMMIT messages to U1 through Un. These cause the transaction to be locally committed.

Phase 4b: C waits for Guaranteed Delivery acknowledgement of these COMMIT messages, and then destroys the backup processes. This represents the successful completion of the transaction.

Each backup is always in one of two states: the abort state or the commit state. Its state is determined by the following transition rules: when created, a backup process enters the abort state; receipt of a COMMIT

message causes it to enter the commit state; receipt of an ABORT message (discussed below) causes it to return to the abort state. The backup process state corresponds exactly to the desired global effect to be achieved. Thus, if a backup process assumes control when it is in the abort state, it should send ABORT messages to all of U1 through Un; if it is in the commit state, it should send COMMIT messages instead.

The operation of a backup process can then be described as follows:

1. When created, a backup sends a message to C, confirming its creation. It also issues a failure watch against its predecessor.

2. Upon receipt of a COMMIT or ABORT message, it acknowledges and makes the appropriate state transition.

3. When notified of the failure of the process it is watching:

    a. If it is watching C, the backup assumes control of the transaction. It issues COMMITs or ABORTs to all the cohorts, depending on its state.

b. If it is watching another backup, it issues a failure watch against that backup's predecessor.

4. If the site on which the backup runs should crash, then the backup ceases to participate in this activity. I.e., the backup process is not continued upon site recovery.

The foregoing works correctly so long as the backup that assumes control does not crash in the midst of sending the COMMIT or ABORT messages to U1, ... Un. If it were to so crash, there is a danger that the next backup to assume control would be in a different state than the failing backup. In such a case, some update process might receive both a COMMIT and an ABORT message. To avoid this problem, each backup process, when created, will be informed of the identity of its higher-numbered backups, which represent the sites that might assume control from it. Then before performing any backup activity, a backup process will ensure that these higher-numbered backups are in the same state that it is in.

Specifically, upon assuming control, a backup process, B, performs the following two-phase procedure:

Phase B1a: B issues COMMIT or ABORT messages (depending on its current state) to all higher-numbered backup processes.

Phase B1b: B waits for each of these backups to acknowledge receipt of the message. Crashsite is called against any backup that does not acknowledge within the time-out period. (This is to ensure that any backup that is up has received the message.)

Phase B2a: B issues COMMIT or ABORT messages (depending on its current state) to all of U1 through Un.

Phase B2b: B waits for Guaranteed Delivery acknowledgement of these messages, and then destroys all the backup processes.

By following this procedure, B insures that, before it ever sends out any messages to the updating processes, all potential take over processes are in the same state that it is in. Therefore, it will not be possible for an updating site to receive both a COMMIT and an ABORT.

3.5.7 Catastrophe In Commit

The algorithm presented in the preceding section will succeed so long as at least one member of the set of the sites {C, B1,....,Bn} remains UP throughout the 4-phase procedure. If this condition does not hold, a commit catastrophe occurs. This catastrophe is not automatically detected by the Relnet. The effect of the catastrophe, however, is simply that some or all of the updates will not be installed. This may force other transactions to wait indefinitely for the completion of the suspended transaction, but it will not produce an inconsistent database. It is left as the responsibility of a system administrator to detect the commit catastrophe and manually issue COMMIT or ABORT messages as appropriate. The likelihood of a commit catastrophe can be lessened by using additional or more stable backup commit sites.

3.6 Conclusion

In this section, we have set forth the basic functionality and architecture of the RelNet, and described implementation mechanisms for its most important facilities. A number of the individual algorithms (such as those for guaranteed delivery and transaction control) should be transportable to other contexts. The Global Time Layer, on the other hand, describes a facility, based on a global clock, which may be specific to the needs of SDD-1. However, we believe that this notion of a uniform clock as a means for coordinating a distributed activity possesses numerous attractive features and may serve as the basis for further developments in distributed system design.

References

[ALSBERG and DAY]
Alsberg, P.A.; and Day, J.D. "A Principle for Resilient Sharing of Distributed Resources", Report from the Center for Advanced Computation, University of Illinois at Urbana-Champaign, Urbana Illinois, 1976.

[BERNSTERN and GOODMAN]
Bernstein, P.A.; and Goodman, N., "Approaches to Concurrency Control in Distributed Database Systems", Proc. 1979 AFIPS National Computer Conference, vol. 48, AFIPS Press, Montvale, N.J.

[BERNSTEIN and SHIPMAN a]
Bernstein, P.A. and D. Shipman; "Concurrency Control in SDD-1: A System for Distributed Databases; Part II: Analysis of Correctness"; ACM Transactions on Database Systems, to appear.

[BERNSTEIN and SHIPMAN b]
Bernstein, P.A. and D.W. Shipman, "A Formal Model of Concurrency Control Mechanisms for Database Systems," Proc. 1978 Berkeley Workshop on Distributed Databases and Computer Networks.

[BERNSTEIN et al. a]
Bernstein, P.A., Rothnie, J.B., Goodman, N., Papadimitriou, C.A.; "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Trans. on Soft. Eng., Vol. SE-4, No. 3 (May 1978), pp. 154-168.

[BERNSTEIN et al. b]
Bernstein, P.A.; Shipman, D.W.; Rothnie, J.B. "Concurrency Control in SDD-1: A System for Distributed Databases; Part I: Description"; ACM Transactions on Database Systems, to appear.

[BERNSTEIN et al. c]
Bernstein, P.A., D.W. Shipman, and W.S. Wong, "Formal Aspects of Serializability in Database Concurrency Control", IEEE Trans. on Soft. Eng., Vol. SE-5, No. 3 (May 1979), pp. 203-215.

[CCA a]

Computer Corporation of America, A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 1, Technical Report No. CCA-77-06, Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[CCA b]
Computer Corporation of America, A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 2, Technical Report No. CCA-78-03, Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[CCA c]
Computer Corporation of America, A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 3, Technical Report No. CCA-78-10, Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[CCA d]
Computer Corporation of America, A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 4, Technical Report No. CCA-79-12, Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[CCA e]
Computer Corporation of America, Datacomputer Version 5 User Manual, Cambridge, Massachusetts, July 1978.

[ESWARAN et al]
Eswaran, K.P.; Gray, J.N.; Lorie, R.A.; Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11 (November 1976), pp. 624-633.

[GOODMAN et. al.]
Goodman, N., P.A. Bernstein, C.L. Reeve, J.B. Rothnie, and E. Wong, "Query Processing in SDD-1: A System for Distributed Databases", submitted for publication.

[GRAY]
Gray, J.N., "Notes on Data Base Operating Systems," Operating Systems: An Advanced Course,

Volume 60 of Lecture Notes in Computer Science, Springer-Verlag, 1978, pp. 393-481.

[HAMMER and SHIPMAN]
Hammer, M.M.; and Shipman, D.W., " The Reliability Mechanisms of SDD-1: A System for Distributed Databases", submitted for publication.

[LAMPORT, L.]
"Time, Clocks and Ordering of Events in a Distributed System", Massachusetts Computer Associates Report #CA-7603-2911, March, 1976. Also submitted to CACM.

[PAPADIMITRIOU et al]
Papadimitriou, C.A.; Bernstein, P.A.; and Rothnie, J.B., "Some Computational Problems Related to Database Concurrency Control", Conference on Theoretical Computer Science, University of Waterloo, Waterloo Ontario, August 1977.

[REED]
Reed, D.P., Naming and Synchronization in a Decentralized Computer System, Ph.D. Thesis, M.I.T., Sept. 1978.

[ROSENKRANTZ et al]
Rosenkrantz, D.J.; Stearns, R.E.; and Lewis, P.M. "System Level Concurrency Control for Distributed Database Systems", ACM Trans. on Database Systems, Vol. 3, No. 2 (June 1978), pp. 178-198.

[ROTHNIE and GOODMAN]
Rothnie, J.B.; and Goodman, N. "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977.

[ROTHNIE et al.]
J. B. Rothnie, Jr., P. A. Bernstein, S. Fox, N. Goodman M. Hammer, T. A. Landers, C. Reeve, D. Shipman, E. Wong "SDD-1: A System for Distributed Databases", ACM Transactions on Database Systems, to appear.

[THOMAS]
Thomas, R.H. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Trans. on Database Systems, Vol. 4, No. 2 (June 1979), pp. 180-209.

[WONG]

Wong, E., "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases", Proc. 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley, California, May 1977.